BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

US006189065B1

(12) **United States Patent**
Arndt et al.

(10) **Patent No.: US 6,189,065 B1**
(45) **Date of Patent: Feb. 13, 2001**

(54) **METHOD AND APPARATUS FOR INTERRUPT LOAD BALANCING FOR POWERPC PROCESSORS**

(75) Inventors: **Richard Louis Arndt**; **Wen-Tzer Thomas Chen**, both of Austin, TX (US)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY (US)

(*) Notice: Under 35 U.S.C. 154(b), the term of this patent shall be extended for 0 days.

(21) Appl. No.: **09/161,622**

(22) Filed: **Sep. 28, 1998**

(51) **Int. Cl.**[7] **G06F 13/24**
(52) **U.S. Cl.** **710/260**; 709/105
(58) **Field of Search** 710/260–269; 709/102–108

(56) **References Cited**

U.S. PATENT DOCUMENTS

3,665,404	*	5/1972	Werner	710/262
4,080,649	*	3/1978	Calle et al.	710/48
4,831,518	*	5/1989	Yu et al.	714/4
4,833,598	*	5/1989	Imamura et al.	710/260
5,179,707	*	1/1993	Piepho	710/260
5,265,215	*	11/1993	Fukuda et al.	710/123
5,423,049	*	6/1995	Kurihara	710/262
5,689,713	*	11/1997	Normoyle et al.	710/263
5,881,293	*	3/1999	Olarig et al.	710/260
5,918,057	*	6/1999	Chou et al.	710/260

OTHER PUBLICATIONS

IBM Technical Disclosure Bulletin, "Balanced Handling of I/O Interrupts in a Multiprocessor System", vol. 36 No. 02 Feb. 1993, pp. 165–166.*

IBM Technical Disclosure Bulletin, D. Giroir et al., "Interrupt Dispatching Method for Multiprocessing System", vol. 27 No. 4B Sep. 1984, pp. 2356–2359.*

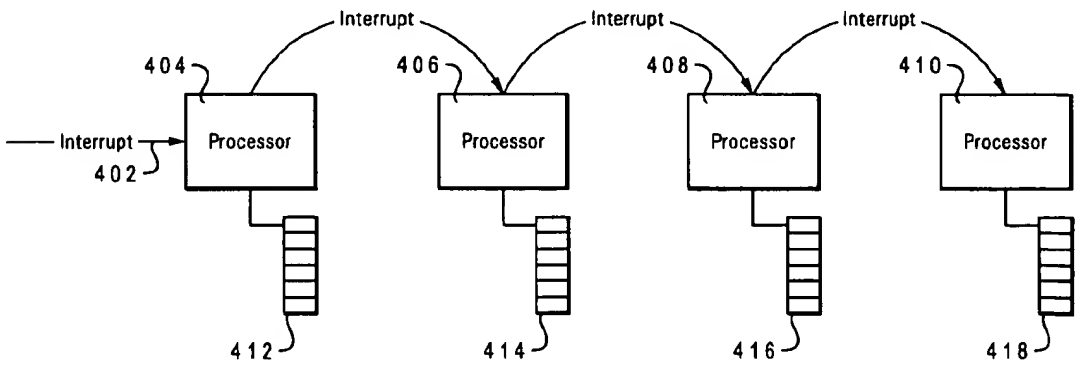
* cited by examiner

Primary Examiner—Sumati Lefkowitz
(74) *Attorney, Agent, or Firm*—Leslie A. Van Leeuwen; Felsman, Bradley, Vaden, Gunter & Dillon, LLP

(57) **ABSTRACT**

Interrupts from an I/O subsystem are first directed to a single processor in a multiple superscalar processor data processing system. If an interrupt load on the processor is sufficiently high, the interrupt is sent (offloaded) to a second specific processor. The process continues throughout all superscalar processors in the data processing system and each processor builds interrupt prediction data corresponding to the interrupt load. A threshold counter may be added to the logic so offloading does not take place until a specified number of interrupts are queued within that specific processor, thus providing a fixed level of prediction data. Some processors may be left out of the offload string so they are not disturbed by an interrupt.

17 Claims, 5 Drawing Sheets



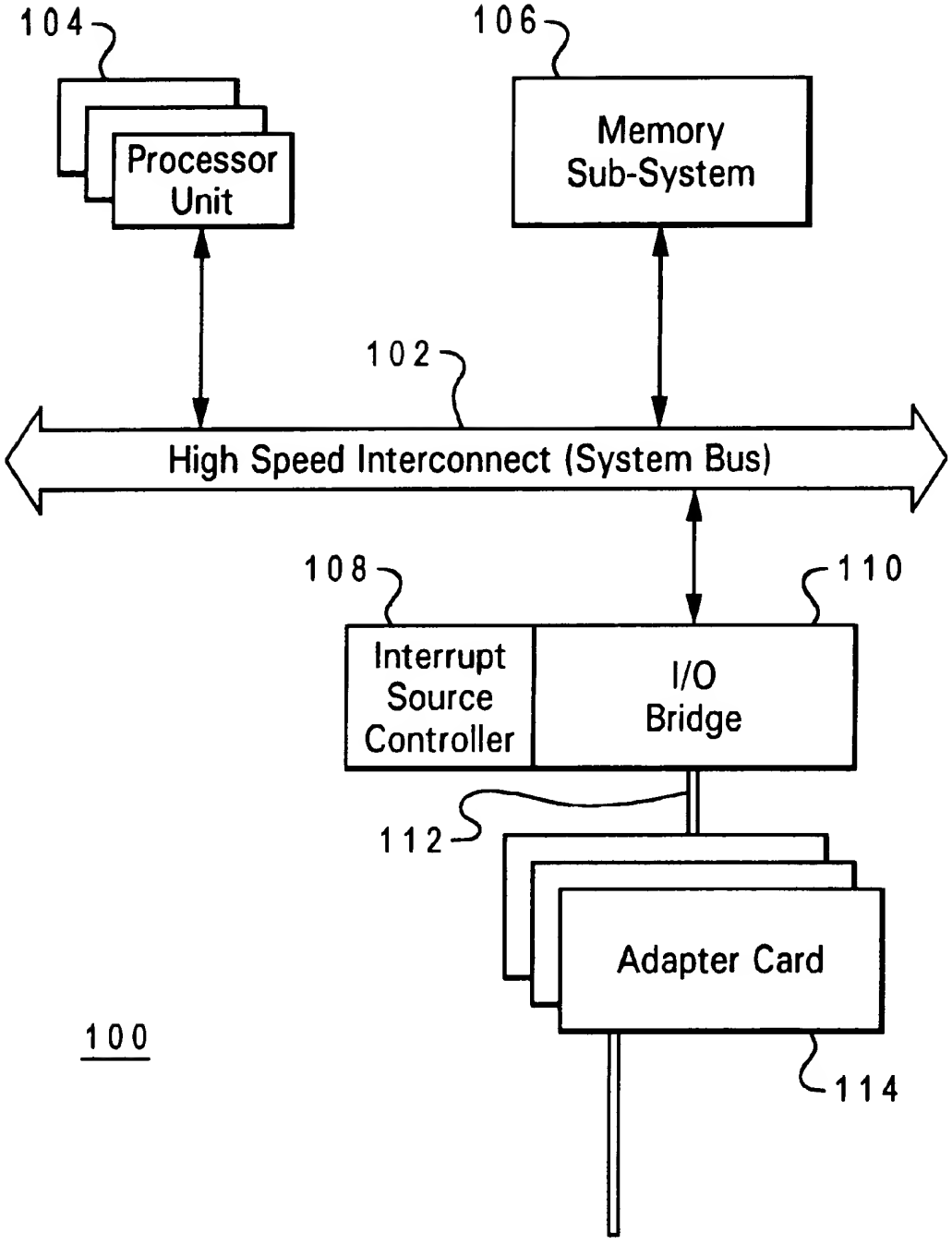

104
106
Processor Unit
Memory Sub-System
102
High Speed Interconnect (System Bus)
108
110
Interrupt Source Controller
I/O Bridge
112
Adapter Card
100
114

Fig. 1

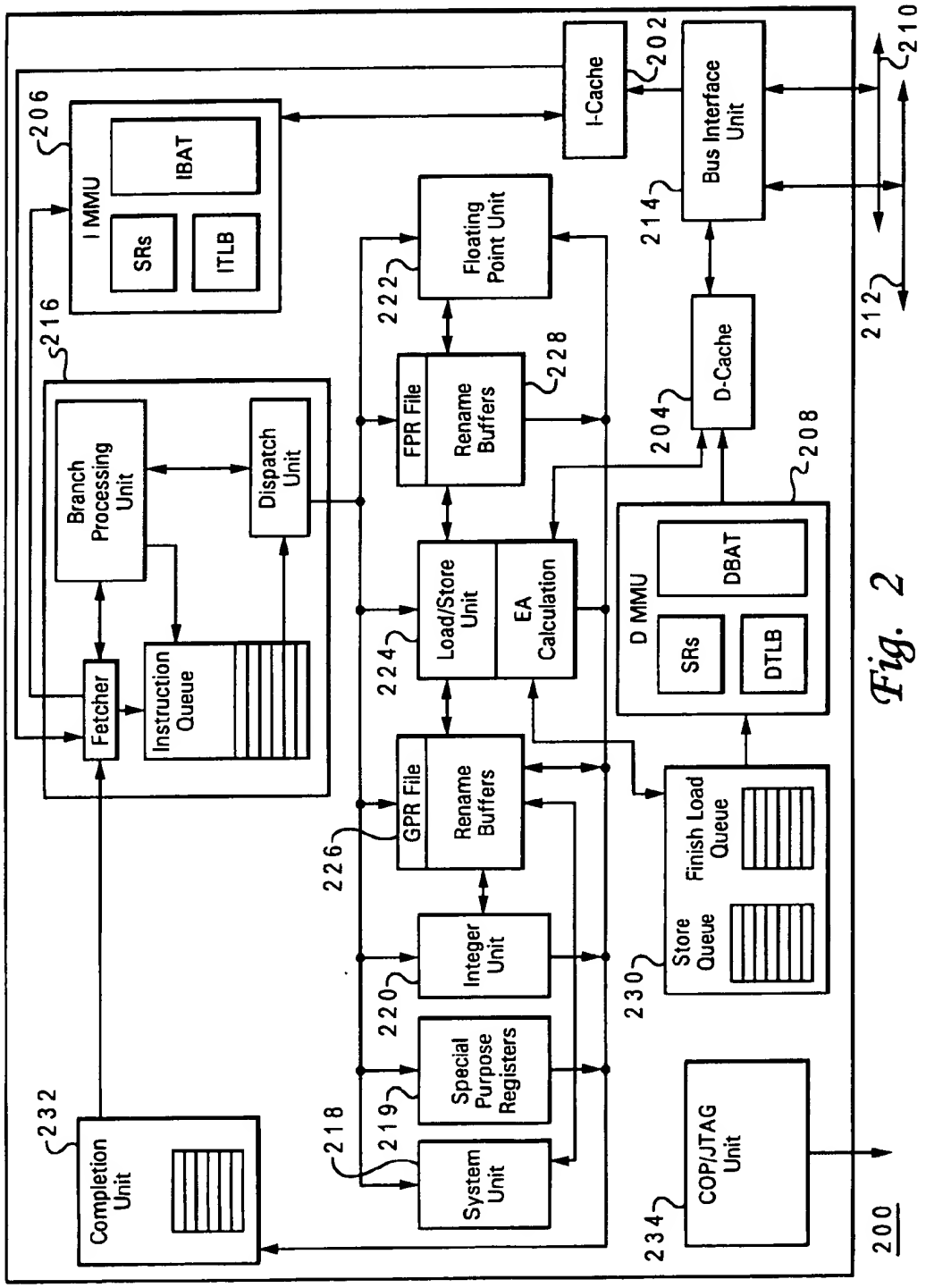

I-Cache
202
Bus Interface Unit
210
206
I MMU
IBAT
SRs
ITLB
214
Floating Point Unit
222
212
216
FPR File
Rename Buffers
228
D-Cache
204
Branch Processing Unit
Dispatch Unit
208
D MMU
DBAT
SRs
DTLB
Load/Store Unit
EA Calculation
224
Fetcher
Instruction Queue
GPR File
Rename Buffers
226
Finish Load Queue
Store Queue
230
Integer Unit
220
Special Purpose Registers
219
218
System Unit
232
Completion Unit
COP/JTAG Unit
234
200

Fig. 2

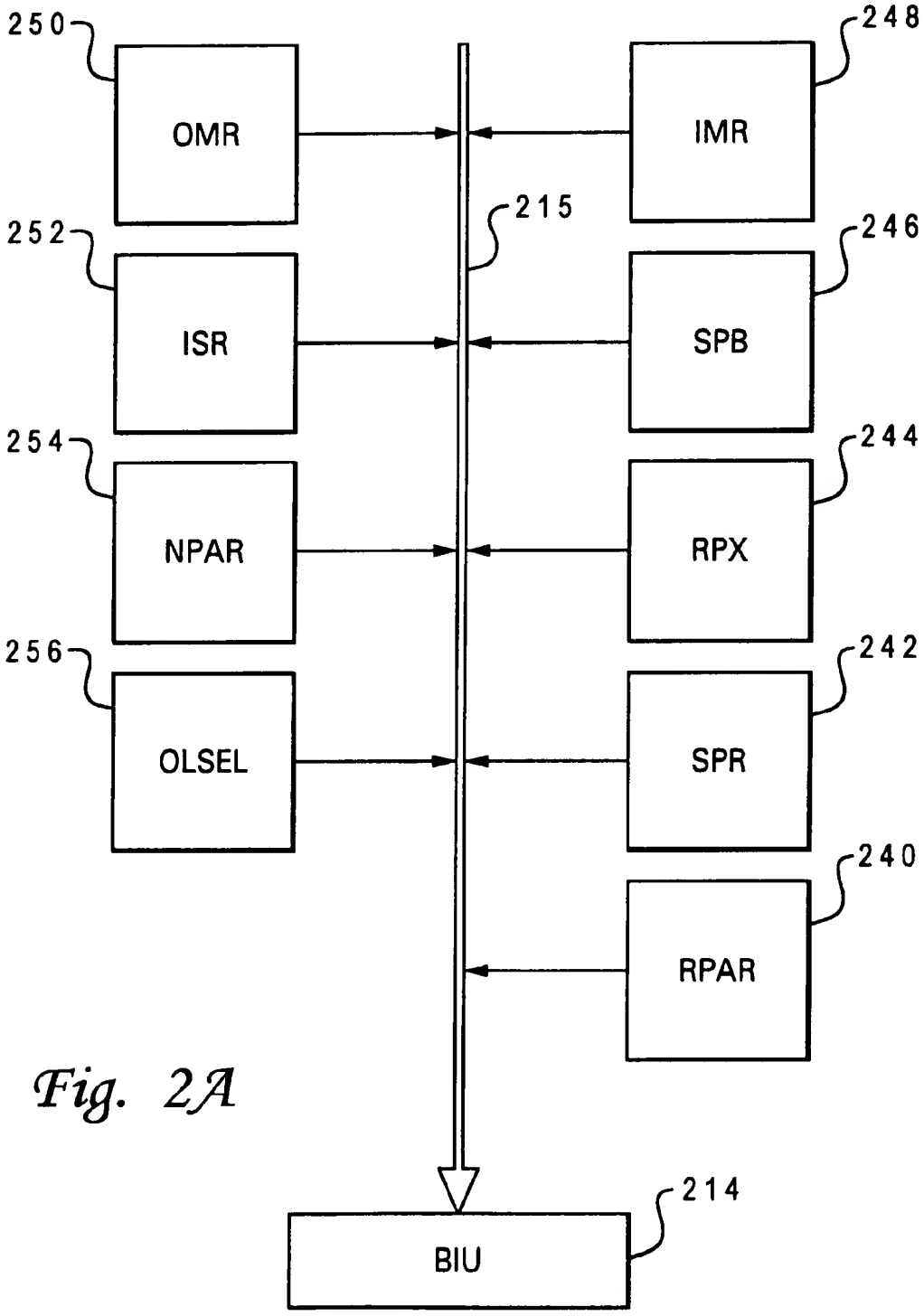

250
OMR
248
IMR
215
252
ISR
246
SPB
254
NPAR
244
RPX
256
OLSEL
242
SPR
240
RPAR
214
BIU

Fig. 2A

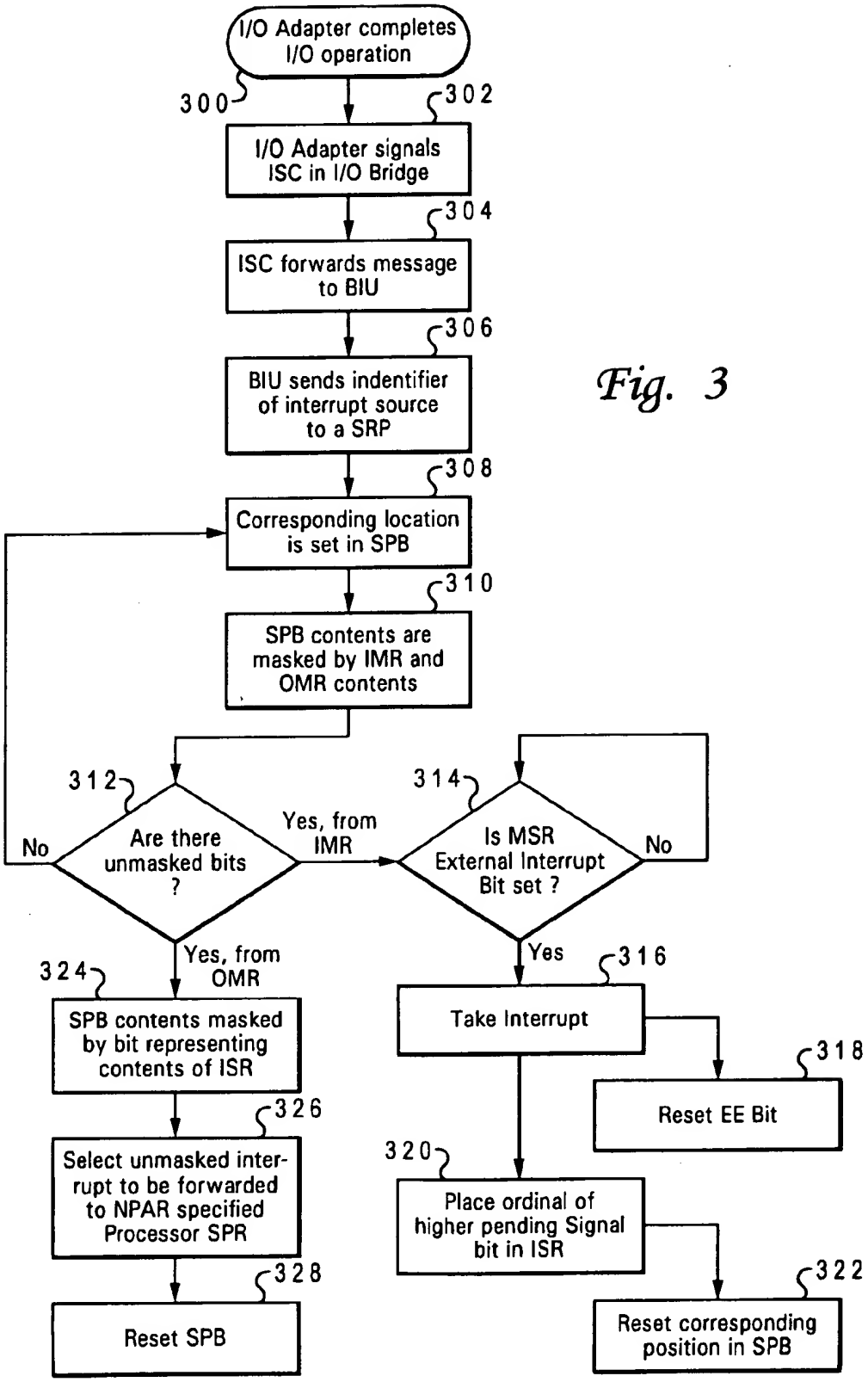

I/O Adapter completes I/O operation
300
302
I/O Adapter signals ISC in I/O Bridge
304
ISC forwards message to BIU
306
BIU sends indentifier of interrupt source to a SRP
308
Corresponding location is set in SPB
310
SPB contents are masked by IMR and OMR contents
312
Are there unmasked bits ?
No
Yes, from IMR
314
Is MSR External Interrupt Bit set ?
No
Yes, from OMR
324
SPB contents masked by bit representing contents of ISR
326
Select unmasked interrupt to be forwarded to NPAR specified Processor SPR
328
Reset SPB
Yes
316
Take Interrupt
318
Reset EE Bit
320
Place ordinal of higher pending Signal bit in ISR
322
Reset corresponding position in SPB

Fig. 3

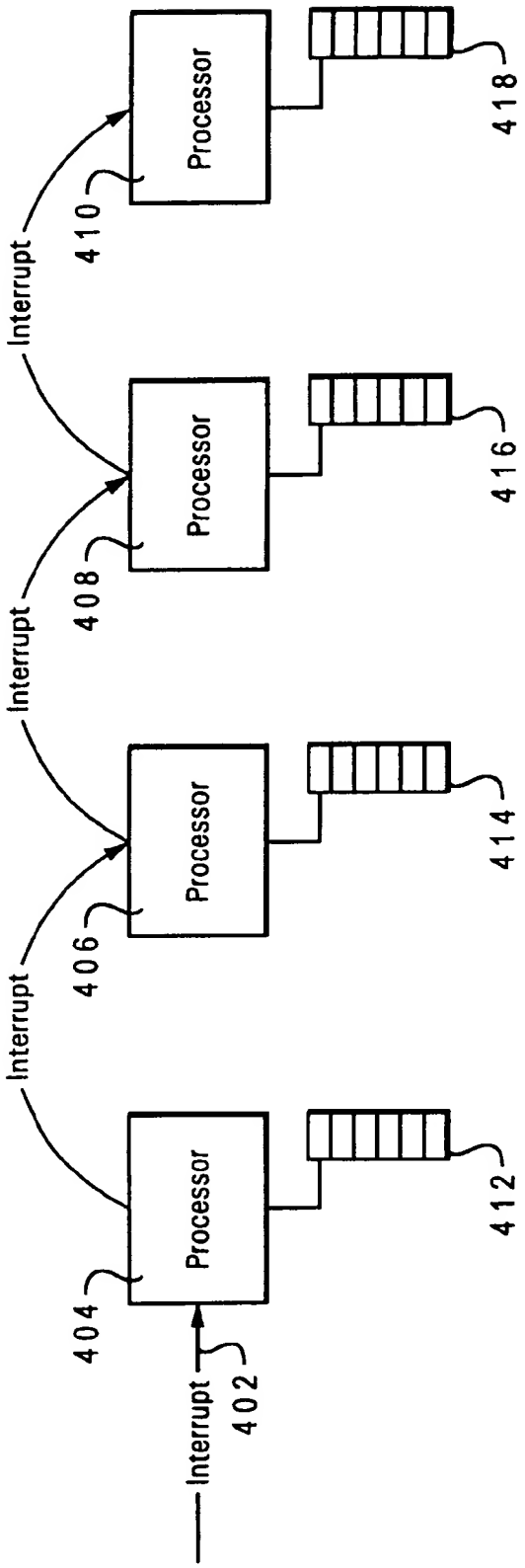

Processor
Processor
Processor
Processor
Interrupt
Interrupt
Interrupt
Interrupt
402
404
406
408
410
412
414
416
418

Fig. 4

METHOD AND APPARATUS FOR INTERRUPT LOAD BALANCING FOR POWERPC PROCESSORS

BACKGROUND OF THE INVENTION

1. Technical Field

The present invention relates in general to data processing systems and in particular to data processing systems utilizing multiple superscalar processors. More particularly, the present invention relates to an interrupt source controller and assignment of interrupts to multiple superscalar processors utilized by the data processing system.

2. Description of the Related Art

An interrupt is an independent signal generated anywhere in a data processing system and is a notification to the processor of the occurrence of an event. External interrupts are usually generated by devices or sub-systems connected to the data processing system. In the case of external interrupts, a signal may be generated by striking a key on the keyboard, depressing a mouse button or a signal from the printer that the printer is active. The interrupt generally has no correlation with the execution of a program, because it originates outside the program. It may occur at any time during the execution of instructions, but is latched inside the processor to be addressed when the active instruction finishes execution. The occurrence of an interrupt is a significant event in the operation of a modern, high speed processor.

Much of the processor's ability to execute at maximum speed comes from the fact that the processor may predict what it will have to do next. This prediction capability is based on the processor's recent past operations. When an interrupt is taken by the processor, the operational context is changed and much of the data used to make operational predictions may become invalid. This may significantly slow down the processor.

Interrupts are usually maskable or non-maskable. Maskable interrupts may be suppressed by an interrupt flag that is placed in the status register referencing a particular interrupt or group of interrupts. However, non-maskable interrupts are typically priority interrupts that must be serviced immediately.

Data processing systems utilizing multiple superscalar processors have a significantly higher interrupt rate than prior art systems. Prior art interrupt source controllers distribute interrupts to the processors in a multiple processor system utilizing one of the following methods:, randomly assign interrupts to one of the processors, assign the interrupt to one specific processor or notify all processors in the system.

Randomly assigning an interrupt to any one of the multiple processors provides a uniform probability of finding a processor without accumulated interrupt data. The lack of consistent and predictable interrupt data on all the processors restricts the efficiency of the random assignment method.

Assigning interrupts to one specific processor insures the best probability of that processor being able to accurately predict operations in the interrupt context. However, as the interrupt load increases, the single processor method may become a bottleneck in the system and load balancing becomes a problem.

Assigning the interrupt to all processors maximally disrupts the system. All processors are interrupted and it must be determined which processor will actually proceed to service the interrupt condition.

In reality, software in these systems mask off interrupts in most processors so that they effectively work as systems in which the interrupts are directed to only one processor with the associated load balance problems. As should thus be apparent, it would be desirable to provide a method that would allow an interrupt source controller in a multiple processor data processing system to service external interrupts promptly, provide a relatively predictable interrupt assignment scheme and improve interrupt load balancing.

SUMMARY OF THE INVENTION

It is therefore one object of the present invention to provide a method and system that will assign external interrupts to one of multiple superscalar processors, in a data processing system, in a relatively predictable manner.

It is another object of the present invention to provide a method and system that will assign multiple external interrupts to succeeding superscalar processors, in a multiple processor data processing system, as each processor reaches a pre-determined interrupt load.

It is a further object of the present invention to provide a method and system that will improve load balancing of the interrupts between processors.

It is yet another object of the present invention to provide a method and system that will direct interrupts to more than one processor.

It is a further object of the present invention to provide a method and system that will offload interrupts from a given processor as a limit is reached.

The foregoing objects are achieved as is now described.

Interrupts from an I/O subsystem are first directed to a first processor in a multiple superscalar processor data processing system. If an interrupt load on the processor is sufficiently high, the interrupt is sent (offloaded) to a second pre-determined processor. The process continues throughout all superscalar processors in the system and each processor builds interrupt prediction data corresponding to the interrupt load on the individual processor. A threshold counter may be added to the logic so offloading does not take place until a specified number of interrupts are queued within that specific processor, thus providing a pre-determined level of prediction data. Some processors may be left out of the offload string so they are not disturbed by an interrupt.

The above as well as additional objects, features, and advantages of the present invention will become apparent in the following detailed written description.

BRIEF DESCRIPTION OF THE DRAWINGS

The novel features believed characteristic of the invention are set forth in the appended claims. The invention itself however, as well as a preferred mode of use, further objects and advantages thereof, will best be understood by reference to the following detailed description of an illustrative embodiment when read in conjunction with the accompanying drawings, wherein:

FIG. 1 depicts a high-level block diagram of a data processing system utilizing multiple superscalar processors, which may be utilized to implement the method and system of the present invention;

FIG. 2 is a block diagram of a processor and related portions of a data processing system in which a preferred embodiment of the present invention may be implemented;

FIG. 2A depicts a high-level block diagram of Special Purpose Registers mapped in accordance with a preferred embodiment of the present invention;

FIG. 3 illustrates a high-level flow chart of a method for reducing processing overhead for high frequency interrupts in a data processing system utilizing multiple superscalar processors, in accordance with a preferred embodiment of the present invention; and

FIG. 4 depicts a high-level block diagram of multiple processors within a data processing system in which a preferred embodiment of the present invention may be implemented.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

With reference now to the figures, and in particular with reference to FIG. 1, a high-level block diagram of a data processing system utilizing multiple superscalar processors, in which a preferred embodiment of the present invention may be implemented, is depicted. Data processing system **100** comprises a high speed interconnect ("System Bus") **102**, interconnected with: one or more superscalar processing units ("processor") **104**, one or more memory subsystems **106**, one or more I/O bridges **110**, each connected to one or more I/O buses **112**, each bus being connected to one or more I/O adapters **114**. Interrupt Source controller **108** is contained within I/O bridge **110**. I/O bridge **110** translates I/O adapter requests into memory operations, processor load/store commands into I/O bus operations and I/O adapter signals into processor interrupts. As is known in the art, an interrupt is a signal generated in a data processing system and is a notification to the processor of the occurrence of an event. External interrupts are generated, for instance, by striking a key on a keyboard or depressing a mouse button and are usually transmitted to a processor via system buses.

Referring now to FIG. 2, a block diagram of a processor and related portions of a data processing system in which a preferred embodiment of the present invention may be implemented are illustrated. Processor **200** is a single integrated circuit superscalar microprocessor, such as the PowerPC™ processor available from IBM Corporation of Armonk, N.Y. Accordingly, processor **200** includes various units, registers, buffers, memories, and other sections, all of which are formed by integrated circuitry. Processor **200** also operates according to reduced instruction set computing ("RISC") techniques.

Processor **200** includes level one (L1) instruction and data caches ("I Cache" and "D Cache") **202** and **204**, respectively, each having an associated memory management unit ("I MMU" and "D MMU") **206** and **208**. As shown in FIG. 2, processor **200** is connected to system address bus **210** and to system data bus **212** via bus interface unit **214**. Instructions are retrieved from system memory (not shown) to processor **200** through bus interface unit **214** and are stored in instruction cache **202**, while data retrieved through bus interface unit **214** is stored in data cache **204**. Instructions are fetched as needed from instruction cache **202** by instruction unit **216**, which includes instruction fetch logic, instruction branch prediction logic, an instruction queue and a dispatch unit.

The dispatch unit within instruction unit **216** dispatches instructions as appropriate to executions units such as system unit **218**, integer unit **220**, floating point unit **222**, or load/store unit **224**. System unit **218** executes condition register logical, special register transfer, and other system instructions.

Special Purpose Registers **219** serve a variety of functions, such as providing controls, indicating status, configuring the processor and performing special operations. Integer or "fixed-point" unit **220** performs add, subtract, multiply, divide, shift or rotate operations on integers, retrieving operands from and storing results in integer or general purpose registers ("GPR File") **226**. Floating point unit **222** performs single precision and/or double precision multiply/add operations, retrieving operands from and storing results in floating point registers ("FPR File") **228**.

Load/store unit **224** loads instruction operands from data cache **204** into integer or floating point registers **226** or **228** as needed, and stores instructions results when available from integer or floating point registers **226** or **228** into data cache **204**. Load and store queues **230** are utilized for these transfers from data cache **204** to and from integer or floating point registers **226** or **228**. Completion unit **232**, which includes reorder buffers, operates in conjunction with instruction unit **216** to support out-of order instruction processing, and also operates in connection with rename buffers within integer and floating point registers **226** and **228** to avoid conflict for a specific register for instruction results. Common on-chip processor ("COP") and joint test action group ("JTAG") unit **234** provides a serial interface to the system for performing boundary scan interconnect tests.

The architecture depicted in FIG. 2 is provided solely for the purpose of illustrating and explaining the present invention, and is not meant to imply any architectural limitations. Those skilled in the art will recognize that many variations are possible. Processor **200** may include, for example, multiple integer and floating point execution units to increase processing throughput. All such variations are within the spirit and scope of the present invention.

Referring now to FIG. 2A, a high-level block diagram of Special Purpose Registers mapped in accordance with a preferred embodiment of the present invention, is illustrated. Each of the illustrated registers is mapped into one or more Special Purpose Registers ("SPR") **219** in processor **200**. The mapping logic is added to the standard logic of processor **200**, including its Bus Interface Unit **214**. This additional logic allows external interrupts from either an interrupt source controller or by one of the system components.

System components external to processor **200** (i.e., I/O bridges, etc.) signal interrupts by writing an interrupt message to Signal Receive Port ("SRP") **242** in a given processor (not shown). The address of the given processor's SRP **242** is supplied to the processor's Bus Interface Unit ("BIU") **214** by the given processor's Receive Port Address Register ("RPAR") **240**. This allows each processor in a multi-processor system to be separately addressed for the purpose of receiving interrupt messages.

Signal Receive Port **242**, receives an interrupt message directed to processor **200** through bus interface unit **214** via bus **215**. The address of the interrupt source received as part of the interrupt message is then placed into Signal Receive Port **242**. Receive Port Expander ("RPX") **244** expands contents of SRP **242** to set a corresponding bit location in Signal Pending Buffer ("SPB") **246**, where it is held for masking purposes.

SPB **246** contents are masked by contents of Interrupt Mask Register ("IMR") **248** to determine interrupt priority. Contents of Offload Mask Register ("OMR") **250** also mask SPB **246**. Contents of these registers and the masking procedure combine and result in the determination of interrupts queued in this processor. The address of the highest priority queued interrupt is kept in the Interrupt Source Register **252**. Next Processor Register ("NPAR") **254** specifies the processor next in line to receive an interrupt and

Offload Selector ("OLSEL") **256** selects an interrupt to be forwarded to the SRP of the processor specified in NPAR **254**.

Referring to FIG. **3**, a high-level flow chart of a method for reducing processing overhead for high frequency interrupts in a data processing system utilizing multiple superscalar processors, in accordance with a preferred embodiment of the present invention, is illustrated. The process begins with step **300**, which depicts an I/O adapter completing an I/O operation. The process proceeds to step **302**, which illustrates the I/O Adapter signaling Interrupt Source Controller in the I/O Bridge. The process proceeds next to step **304**, which depicts the Interrupt Source Controller writing a message containing the interrupt source identifier of the I/O Adapter to the Bus Interface Unit of a processor (generally, a pre-determined processor in a multiple processor system) to signal an interrupt to the system processor. An external interrupt is permitted from either an interrupt source controller or by one of the system components (such as another Processor or adapter) writing a data value to an addressed location in the processor to be interrupted which is specified in a Receive Port Address Register ("RPAR").

The process proceeds to step **306**, which illustrates the pre-determined processor's Bus Interface Unit ("BIU") responding to store operations directed to an address corresponding to the value contained in the Receive Port Address Register ("RPAR"), by sending a value corresponding to the identifier of the interrupt source into the Signal Receive Port ("SRP"). The process then passes to step **308**, which depicts utilizing contents of the SRP to set the corresponding bit location in the Signal Pending Buffer.

The process proceeds next to step **310**, which illustrates contents of an Interrupt Mask Register ("IMR") masking contents of the Signal Pending Buffer. Additionally, contents of the OMR are used to mask the contents of the SPB. The process then passes to step **312**, which depicts a determination of whether there is a resulting unmasked bit. If there are no unmasked bits, the process returns to step **310** and repeats the steps of the process, waiting for the processor to change the IMR or OMR, or to receive a new interrupt as in step **300**. If there is an unmasked bit, as a result of masking with contents of IMR, the process proceeds to step **314**, which illustrates a determination of whether a Machine State Register External Interrupt Enable ("EE") bit is set. If it is not set, the process returns to step **314** and repeats until the processor is able to accept an interrupt. If, in step **314**, the determination is made that a Machine State Register External Interrupt Enable bit is set, the process then proceeds to step **316**, which depicts the processor receiving the interrupt. Subsequently, the process passes to step **318**, which illustrates the External Interrupt Bit being reset.

After the processor receives the interrupt and resets the External Interrupt Enable bit, the process then proceeds to step **320**, which depicts the ordinal of the higher order masked pending signal bit being placed in the Interrupt Source Register ("ISR"). Subsequently the process passes to step **322**, which illustrates the act of loading the ISR, which causes the corresponding position to be reset in the Signal Pending Buffer.

Returning to step **310** and **312**, concurrently with masking of Signal Pending Buffer by the contents of the Interrupt Mask Register, if there are unmasked bits from an Offload Mask Register, the process proceeds to step **324**. Masking the SPB with the Offload Mask prevents the offloading of certain interrupt sources that for some reason must not be offloaded to another processor, i.e., if the service routine could only run on one processor, the interrupt would not be offloaded. Step **324** depicts the contents of the Signal Pending Buffer being masked by the Offload Mask Register and by the bit represented by the contents of the Interrupt Source Register. The result represents those interrupts queued in the specified processor that are available for offload to another processor. The process next proceeds to step **326**, which illustrates an Offload Selector selecting an interrupt to be forwarded to the Signal Receive Port of a processor specified in a Next Processor Address Register. Step **326** may include a counting function such that interrupts are not offloaded until some pre-determined threshold is reached. This prevents temporary bursts of interrupts from unduly disturbing other processors while allowing offloads as the interrupt workload grows larger. The processor then continues to step **328**, which depicts the offloaded interrupt being forwarded and the corresponding position in the Signal Pending Buffer being reset.

Referring now to FIG. **4**, a high-level block diagram of a method for directing and offloading an interrupt to successive, designated processors in a data processing system. Multi-processor subsystem **400** of data processing system (not shown) comprises: multiple processors **404**, **406**, **408**, and **410** in which interrupts are queued in processor queues **412**, **414**, **416** and **418** and, as is known in the art, there may be more or less processors contained within a multi-processor data processing system. Logic for offloading the interrupts to successive processors may contain a threshold counter for restricting the number of interrupts within each processor's queue. By pre-determining the number of interrupts in a processor's queue, prediction data is more accurate and improves efficiency of the processor.

Interrupt **402** is written to a first processor in a multiprocessor system. Interrupt **402** is received in a Signal Receive Port (not shown) and if the interrupt queue for processor **404** has reached a pre-determined level, the interrupt is offloaded to processor **406**. Depending on the predetermined level for queue **414**, the interrupt may be sent to processor **408**, and so on, until the interrupt is entered into the interrupt queue of one of the processors in the system.

It is important to note that while the present invention has been described in the context of a fully functional data processing system, those skilled in the art will appreciate that the mechanism of the present invention is capable of being distributed in the form of a computer readable medium of instructions in a variety of forms, and that the present invention applies equally, regardless of the particular type of signal bearing media utilized to actually carry out the distribution. Examples of computer readable media include: nonvolatile, hard-coded type media such as read only memories (ROMs) or erasable, electrically programmable read only memories (EEPROMs), recordable type media such as floppy disks, hard disk drives and CD-ROMs, and transmission type media such as digital and analog communication links.

While the invention has been particularly shown and described with reference to a preferred embodiment, it will be understood by those skilled in the art that various changes in form and detail may be made therein without departing from the spirit and scope of the invention.

What is claimed is:

1. A method for servicing an interrupt message in a data processing system having a plurality of processors, comprising the steps of:

mapping special Purpose registers in each of said plurality of processors wherein said mapping logic provides, at least:

a receive port register for holding a unique address for said processor;

a signal pending buffer for holding a bit location;

at least two masking registers for said interrupt message;

a next processor register for designating a successive processor to receive an interrupt;

directing said interrupt message to a first processor;

masking said interrupt message to determine said interrupt message priority;

storing, in an interrupt source register, the address of the highest priority qeued said interrupt message;

utilizing the contents of said signal pending buffer and said at least two masking registers for determining the interrupts that are queued in said first processor;

utilizing an offload selector for offloading a specific (said interrupt message to a processor specified in said next processor register if said first processor is busy servicing another interrupt; and

offloading said interrupt message from each successive busy processor to a different processor specified in said busy processor's next processor register until said interrupt message is accepted by one of said plurality of processors.

2. The method of claim **1**, wherein said step of directing an interrupt signal to a first processor further comprises

transmitting said interrupt message via an interrupt source controller to said processor; and

storing the source address of said interrupt message on board said processor.

3. The method in claim **1**, wherein said step of utilizing said offload selector for offloading a specific said interrupt message to a processor specified in said next processor register if said first processor is busy servicing another interrupt message, further comprises:

pre-determining a specific number of interrupts each processor will manage prior to offloading additional interrupts to a second processor.

4. The method in claim **1**, further comprising storing interrupt prediction data in each processor.

5. The method in claim **1**, further comprising the step of

transmitting said interrupt signal directly from a data processing system component.

6. The method in claim **1**, wherein said step of offloading said interrupt message from each successive busy processor to a different processor specified in said busy processor's next processor register until said interrupt message is accepted by one of said plurality of processors further comprises:

pre-designating a receiving order of said plurality of processors to which interrupt signals will be directed;

limiting a number of interrupt signals that each one of said plurality of processors will manage prior to offloading additional interrupts to a specified second processor, by including a threshhold counter; and

offloading said additional interrupts to each of said plurality of processors in said receiving order.

7. A data processing system having a plurality of processors, comprising:

logic within each processor for mapping special purpose registers wherein said mapped registers comprise, at least:

a receive report register for holding a unique address for said processor;

a signal pending buffer for holding a bit corresponding to an interrupt message address source;

at least two masking registers for masking said interrupt message to determine said interrupt message priority;

a next processor register for designating a next processor in line to receive an interrupt;

means for directing an interrupt message to a first processor;

an interrupt source register for storing the address of the highest priority queued said interrupt message;

an offload selector for offloading said interrupt message to another processor if said first processor is busy servicing another interrupt signal; and

means for offloading said interrupt message from each said busy processor to a processor, specified in said busy processor's next processor register, within said data processing system until said interrupt signal is accepted by one of said plurality of processors.

8. The data processing system of claim **7**, wherein said an offload selector for offloading said interrupt message to a second processor if said first processor is busy servicing another interrupt signal further comprises:

logic means for pre-designating a second processor for receiving an interrupt signal if said first processor is busy; and

logic containing a threshhold counter for predetermining a specific number of interrupts for each processor to manage prior to offloading additional interrupts to said second processor.

9. The data processing system of claim **7**, further comprising

a mapped register onboard said processor for storing interrupt message prediction data.

10. The data processing system of claim **8**, further comprising

a means for receiving an external interrupt message directly from a data processing system component.

11. The data processing system of claim **7**, wherein said means for offloading said interrupt message to each of said plurality of processors until said interrupt message is accepted by one of said processors further comprises:

means for pre-designating a receiving order of said plurality of processors that will receive interrupts;

a threshhold counter for limiting a number of interrupt messages that each one of said plurality of processors will manage prior to offloading additional interrupts to a specified second processor; and

logic means for successively offloading said additional interrupts to each of said plurality of processors.

12. A computer-readable medium for servicing an interrupt message in a data processing system having a plurality of processors, comprising:

instructions within said computer-readable medium for mapping special purpose registers in each of said plurality of processors wherein said mapping logic provides, at least:

a receive port register for holding a unique address for said processor;

a signal pending buffer for holding a bit location:

at least two masking registers for said interrupt message;

a next Processor register for designating a successive processor to receive an interrupt;

instructions within said computer-readable medium for directing said interrupt message to a first processor;

instructions within said computer-readable medium for masking said interrupt message to determine said interrupt message priority;

instructions within said computer-readable medium for storing, in an interrupt source register, the address of the highest priority queued said interrupt message;

instructions within said computer-readable medium for utilizing the contents of said signal pending buffer and said at least two masking registers for determining the interrupts that are queued in said first processor;

instructions within said computer-readable medium for utilizing an offload selector for offloading a specific said interrupt message to a processor specified in said next processor register if said first processor is busy servicing another interrupt; and

instructions within said computer-readable medium for offloading said interrupt message from each successive busy processor to a different processor within said data processing system until said interrupt message is accepted by one of said plurality of processors.

13. The computer-readable medium of claim **12**, wherein said instructions for directing an interrupt message to a first processor further comprises

instructions within said computer-readable medium for transmitting said interrupt message via an interrupt source controller to said processor; and

instructions within said computer-readable medium for storing the source address of said interrupt message on board said first processor.

14. The computer-readable medium of claim **12**, wherein said instructions for offloading said interrupt message if said first processor is busy servicing another interrupt message further comprises:

instructions within the computer-readable medium for predetermining a specific number of interrupts for each processor to manage prior to offloading additional interrupts to a second processor.

15. The computer-readable medium of claim **12**, further comprising

instructions within the computer-readable medium for storing interrupt prediction data in each processor.

16. The computer-readable medium of claim **12**, further comprising

instructions within the computer-readable medium for transmitting said interrupt message directly from a data processing system component.

17. The computer-readable medium of claim **12**, wherein said instructions for offloading said interrupt message to each of said plurality of processors until said interrupt message is accepted by one of said processors further comprises:

instructions within the computer-readable medium for predesignating a receiving order of said plurality of processors that will receive interrupts;

instructions within the computer-readable medium for limiting a number of interrupt messages that each one of said plurality of processors will manage prior to offloading additional interrupts to a specified second processor, by including a threshold counter; and

instructions within the computer-readable medium for successively offloading said additional interrupts to each of said plurality of processors.

* * * * *

US006338133B1

(12) **United States Patent**
Schroter

(10) **Patent No.: US 6,338,133 B1**
(45) **Date of Patent: Jan. 8, 2002**

(54) **MEASURED, ALLOCATION OF SPECULATIVE BRANCH INSTRUCTIONS TO PROCESSOR EXECUTION UNITS**

(75) Inventor: **David Andrew Schroter**, Round Rock, TX (US)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/267,200**

(22) Filed: **Mar. 12, 1999**

(51) **Int. Cl.**[7] **G06F 9/38**; G06F 9/445; G06F 9/312; G06F 9/24

(52) **U.S. Cl.** **712/214**; 712/206; 712/207; 712/239; 712/237; 712/245; 711/137; 711/144; 711/145; 711/123; 711/125; 710/54; 710/56; 710/57

(58) **Field of Search** 712/231, 232, 712/233, 234, 236, 237, 239, 207, 206, 216, 222, 245, 202, 228, 23, 215, 223, 235, 126, 217, 204, 226, 214; 711/137, 144, 145; 710/54, 56, 57

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,228,131	A	7/1993	Ueda et al.	712/240
5,333,283	A	7/1994	Emma et al.	712/236
5,421,020	A	5/1995	Levitan	712/237
5,454,117	A	9/1995	Puziol et al.	712/23
5,463,745	A *	10/1995	Vidwans et al.	712/218
5,564,111	A	10/1996	Glew et al.	174/53
5,611,063	A	3/1997	Loper et al.	712/205
5,613,083	A	3/1997	Glew et al.	711/207
5,613,125	A	3/1997	Nguyen et al.	713/1
5,625,787	A	4/1997	Mahin et al.	712/204
5,634,103	A	5/1997	Dietz et al.	712/235
5,640,526	A	6/1997	Mahin et al.	712/207
5,644,744	A	7/1997	Mahin et al.	712/207
5,664,136	A	9/1997	Witt et al.	712/208
5,724,565	A	3/1998	Dubey et al.	712/245
5,729,728	A	3/1998	Colwell et al.	712/234
5,752,014	A	5/1998	Mallick et al.	712/240
5,778,245	A *	7/1998	Papworth et al.	712/23
6,009,499	A *	12/1999	Koppala	711/132

FOREIGN PATENT DOCUMENTS

EP	0751458 A1	1/1997	

OTHER PUBLICATIONS

IBM Technical Disclosure Bulletin, "Cycle Steal at Rename Register," vol. 38, No. 9, Sep. 1995, pp. 363–364.

IBM Technical Disclosure Bulletin, "Data Restoration of an Addressable Array," vol. 35, No. 1A, Jun. 1992, pp. 222–225.

Pradeep K. Dubey et al., "Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine Grained Multithreading", Conference on Parallel Architectures and Compilation Techniques, 1995, pp. 102–121.

Rainer Mueller, "The MC88110 Instruction Sequencer", Northcon Conference Record, Oct. 19–21, 1992, pp. 198–201.

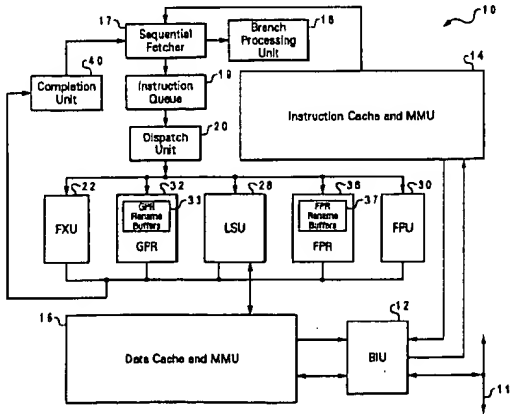
* cited by examiner

Primary Examiner—Daniel H. Pan
(74) *Attorney, Agent, or Firm*—Leslie A. Van Leeuwen; Bracewell & Patterson, L.L.P.

(57) **ABSTRACT**

A method and system for branch dispatching of instructions in a data processor. A processor having one or more buffers for storing instructions and one or more execution units for executing instructions is utilized. Each unit has a corresponding queue which holds instructions pending execution. First, a threshold level (selected maximum number of instructions in the instruction queue) is set. The current utilization measure for one or more execution units in the data processing system is determined. The current utilization measure is compared to the predetermined threshold value; and a speculative branch instruction is dispatched to a selected execution unit when the current utilization measure is less than the predetermined threshold value.

12 Claims, 5 Drawing Sheets



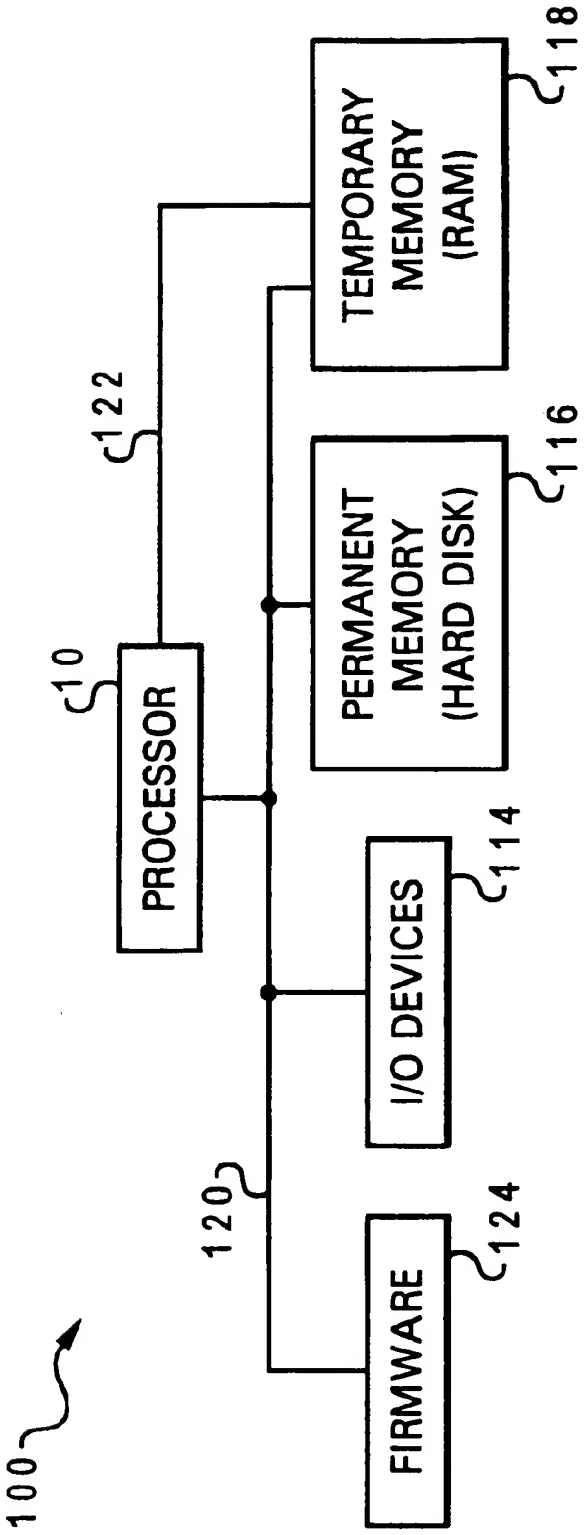

TEMPORARY MEMORY (RAM)
118
122
PERMANENT MEMORY (HARD DISK)
116
10
PROCESSOR
I/O DEVICES
114
120
FIRMWARE
124
100

Fig. 1

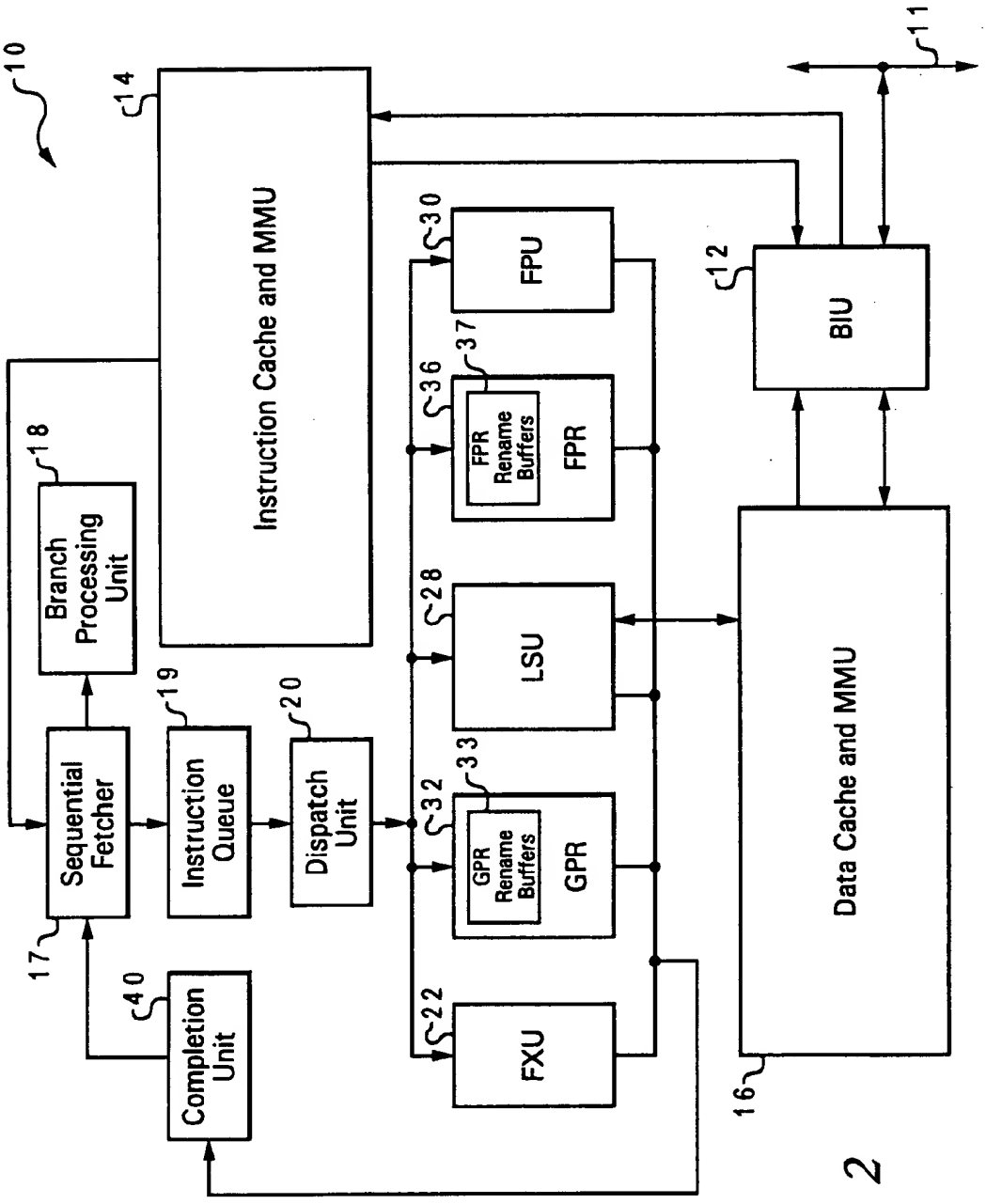

10
11
12
14
16
17
18
19
20
22
28
30
32
33
36
37
40
Instruction Cache and MMU
Branch Processing Unit
Sequential Fetcher
Instruction Queue
Dispatch Unit
Completion Unit
FPU
FPR Rename Buffers
FPR
LSU
GPR Rename Buffers
GPR
FXU
BIU
Data Cache and MMU

Fig. 2

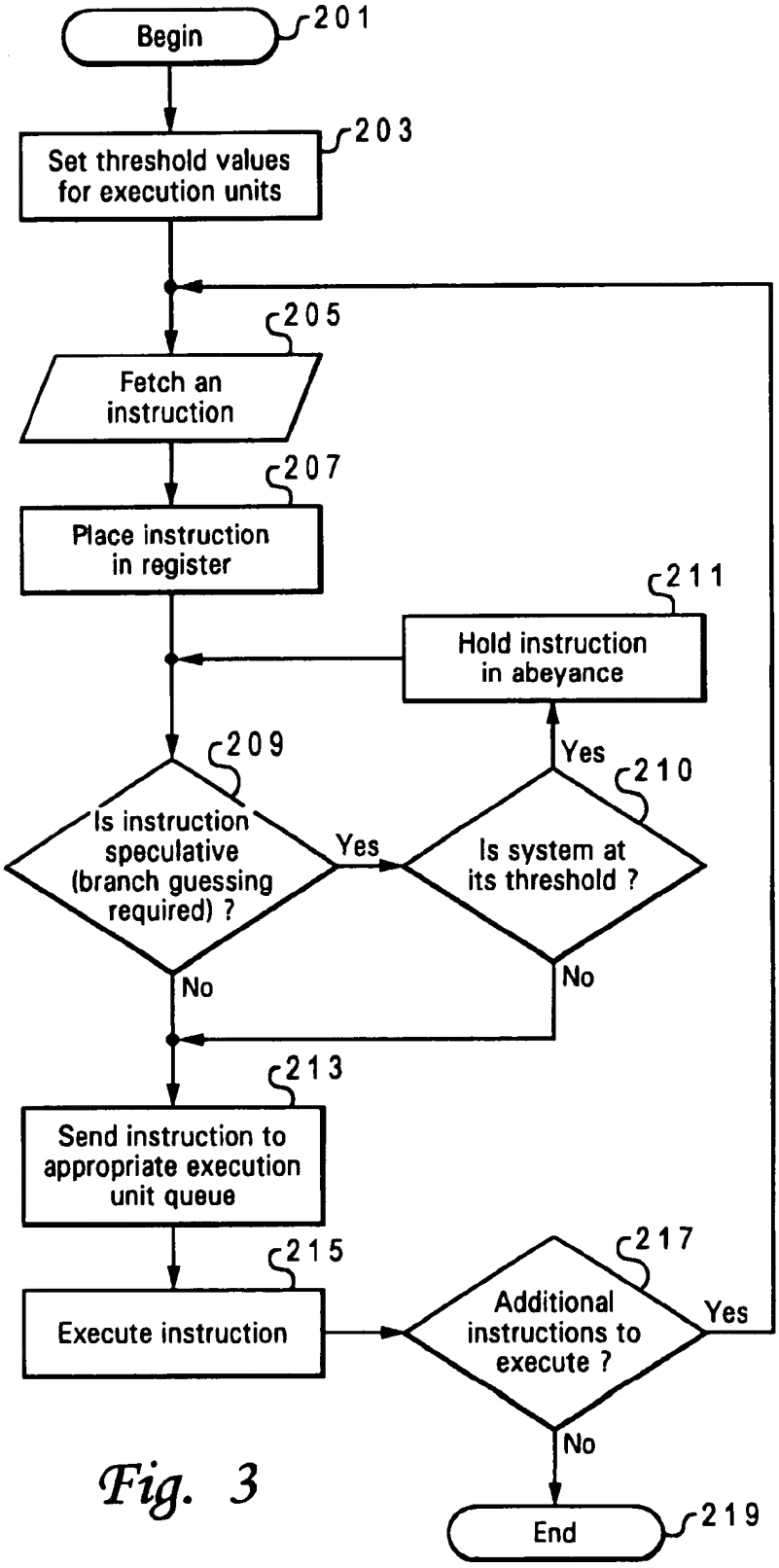

Begin
201
Set threshold values for execution units
203
Fetch an instruction
205
Place instruction in register
207
Hold instruction in abeyance
211
Is instruction speculative (branch guessing required) ?
209
Yes
Is system at its threshold ?
210
Yes
No
No
Send instruction to appropriate execution unit queue
213
Execute instruction
215
Additional instructions to execute ?
217
Yes
No
End
219

Fig. 3

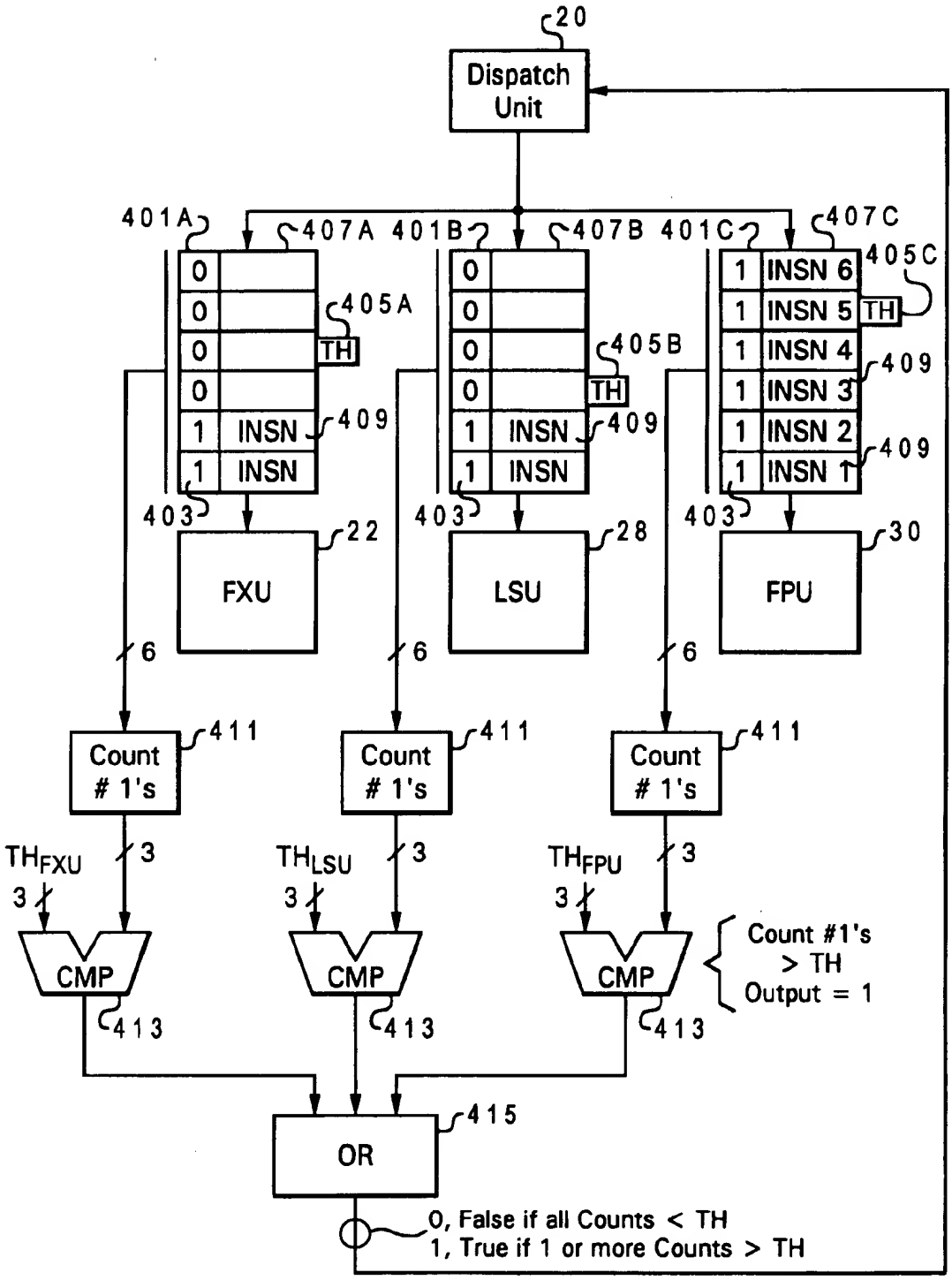

20
Dispatch Unit
401A
407A
401B
407B
401C
407C
405A
405B
405C
TH
409
INSN
INSN 6
INSN 5
INSN 4
INSN 3
INSN 2
INSN 1
403
22
28
30
FXU
LSU
FPU
6
411
Count # 1's
THFXU
THLSU
THFPU
3
CMP
413
Count #1's > TH Output = 1
415
OR
0, False if all Counts < TH
1, True if 1 or more Counts > TH

Fig. 4A

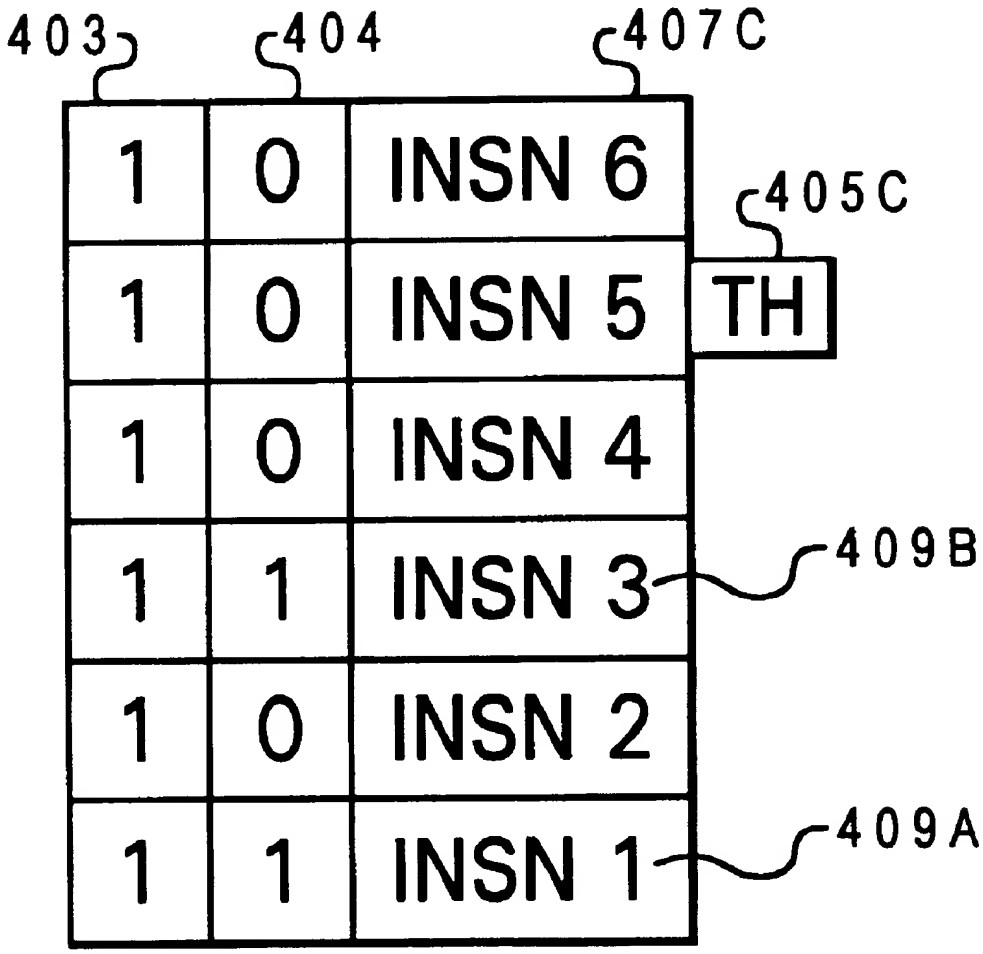

403
404
407C
405C
1
0
INSN 6
1
0
INSN 5
TH
1
0
INSN 4
1
1
INSN 3
409B
1
0
INSN 2
1
1
INSN 1
409A

Fig. 4B

MEASURED, ALLOCATION OF SPECULATIVE BRANCH INSTRUCTIONS TO PROCESSOR EXECUTION UNITS

BACKGROUND OF THE INVENTION

1. Technical Field

The present invention relates in general to a method and system for data processing and in particular to a method and system for executing instructions within a data processor. Still more particularly, the present invention relates to a method and system for executing instructions within a data processor such that speculative branch instructions are controlled to provide more efficient execution.

2. Description of the Related Art

A conventional high performance superscalar processor typically includes an instruction cache for storing instructions, an instruction buffer for temporarily storing instructions fetched from the instruction cache for execution, a number of execution units for executing sequential instructions, a branch processing unit (BPU) for executing branch instructions, a dispatch unit for dispatching sequential instructions from the instruction buffer to particular execution units, and a completion buffer for temporarily storing instructions that have finished execution, but have not been completed.

As is well known in the art, sequential instructions fetched from the instruction queue are stored within the instruction buffer pending dispatch to the execution units. In contrast, branch instructions fetched from the instruction cache are typically forwarded directly to the branch processing unit for execution. In some cases, the condition register value upon which a conditional branch depends can be ascertained prior to executing the branch instruction, that is, the branch can be resolved prior to execution. If a branch is resolved prior to execution, instructions at the target address of the branch instruction are fetched and executed by the processor. In addition, any sequential instructions following the branch that have been pre-fetched are discarded. However, the outcome of a branch instruction often cannot be determined prior to executing the branch instruction due to a condition register dependency. When a branch instruction remains unresolved at execution, the branch processing unit utilizes a prediction mechanism, such as a branch history table, to predict which execution path should be taken. In conventional processors, the dispatch of sequential instructions following a branch predicted as taken is halted and instructions within the speculative target instruction stream are fetched during the next processor cycle. If the branch that was predicted as taken is resolved as mispredicted, a mispredict penalty is incurred by the processor due to the cycle time required to restore the sequential execution stream following the branch instructions.

A high performance processor achieves high instruction throughput by fetching and dispatching instructions under the assumption that branches are correctly predicted and allows instructions to execute without waiting for the completion of previous instructions. This is commonly known as speculative execution, i.e., executing instructions that may or may not have to be executed. The CPU guesses which path the branch was going to take. This guess may be a very intelligent guess (as in a branch history table) or very simple (as in always guess path not taken). Once the guess is made, the CPU starts executing that path. Typically, the processor executes instructions speculatively when it has resources that would otherwise be idle, so that the operation may be done at minimum or no cost. Therefore, in order to

enhance performance, some processors speculatively execute unresolved branch instructions by predicting whether or not the indicated branch will be taken. Utilizing the result of the prediction, the fetcher is then able to fetch instructions within the speculative execution path prior to the resolution of the branch, thereby avoiding a stall in the execution pipeline if the branch is resolved as correctly predicted. If the guess is correct, and there are no holes or delays in the pipeline, execution continues at full speed. If, however, subsequent events indicate that the speculative instruction should not have been executed, the processor has to abandon any result that the speculative instruction produced and begin executing the path that should have been taken. The processor "flushes" or throws away the instruction results, backs itself up to get a new address and executes the correct instruction.

Most operations can be performed speculatively, as long as the processor appears to follow a simple sequential method such as those in a scalar processor. For some applications, however, speculative operations can be a severe detriment to the performance of the processor. For example, in the case of executing a load instruction after a branch instruction (known as speculative load because the load instruction is executed speculatively without knowing exactly which path of the branch would be taken), if the predicted execution path is incorrect, there is a high delay penalty is incurred when the pending speculative load in the instruction stream requests the required data from the system bus. In many applications, the rate of mis-predicted branches is high enough that the cost of speculatively accessing the system bus is prohibitively expensive. Furthermore, essential data stored in a data cache may be displaced by some irrelevant data obtained from the system bus because of a wrongful execution of a speculative load instruction caused by misprediction.

Prior art handling of this speculative execution of instructions includes U.S. Pat. No. 5,454,117 which discloses a branch prediction hardware mechanism. The mechanism performs speculative execution based on the branch history information in a table. However, it does not provide a means for prediction based on the current status of the branch execution unit. Similarly, U.S. Pat. No. 5,611,063 discloses a method for tracking allocation of resources within a processor utilizing a resource counter which has two bits set in two possible states corresponding to whether or not the instruction is speculative or when dispatched to an execution unit respectively.

U.S. Pat. No. 5,752,014 discloses a selection from among a plurality of branch prediction methodologies, namely dynamic prediction and static prediction, in speculative execution of conditional branch instructions. It discusses the execution of the instructions based on the prediction and subsequent conditional branch instruction.

No prior art discloses a method or system for determining whether to dispatch a speculative instruction based on current loading conditions. Consequently, a processor and method for speculatively executing conditional branch instructions are needed which intelligently determines when it is necessary to utilize speculative prediction.

In modern microprocessors, there are many mechanisms known to speculatively execute instructions. Speculative execution can improve performance significantly if the speculation is correct. In speculatively executing branch instructions, prediction means improve the likelihood of guessing the correct path. However, if the guess is wrong recovery means must be utilized to cancel the effect of

instructions that should not be completed. In actual practice, it is sometimes difficult and expensive to selectively cancel instructions as a result of a bad branch speculation. This is especially true in superscalar systems where instructions are executed out-of-order. A new method is needed to better determine when speculative branch instructions are to be dispatched.

It would therefore be desirable to provide a method and system for selectively executing speculative branch instructions in a high performance processor by utilizing a better prediction scheme. It is further desirable to provide a method and system which dispatch speculative instructions only when the system is below a predefined load capacity to prevent unfettered dispatching of speculative instructions.

SUMMARY OF THE INVENTION

It is therefore one object of the present invention to provide an improved data processor.

It is another object of the present invention to provide a method and system for executing instructions within a data processor.

It is yet another object of the present invention to provide a method and system for executing instructions within a data processor such that speculative branch instructions are controlled to provide more efficient execution.

The foregoing objects are achieved as is now described. A method and system is disclosed for speculative branch dispatching of instructions in a data processor. A processor having one or more buffers for storing instructions and one or more execution units for executing instructions is utilized. Each execution unit has a corresponding queue which holds instructions pending execution. First, a threshold level (selected maximum number of instructions in the instruction queue) is set. The current utilization measure for one or more execution units in the data processing system is then determined. The current utilization measure is compared to the predetermined threshold value; and a speculative branch instruction is dispatched to a selected execution unit when the current utilization measure is less than the predetermined threshold value.

In one embodiment of the invention, the branch dispatching check occurs at each unit individually. A comparison is made of the number of instructions queued to the execution unit with its threshold value. The dispatching step dispatches the branch instructions to that execution unit only when the number of instructions queued at the execution unit is lower than the threshold value for that unit.

The above as well as additional objects, features, and advantages of the present invention will become apparent in the following detailed written description.

DESCRIPTION OF THE DRAWINGS

The novel features believed characteristic of the invention are set forth in the appended claims. The invention itself however, as well as a preferred mode of use, further objects and advantages thereof, will best be understood by reference to the following detailed description of an illustrative embodiment when read in conjunction with the accompanying drawings, wherein:

FIG. **1** is a block diagram of a data processing system utilized in one embodiment of the present invention;

FIG. **2** is a block diagram of a preferred embodiment of a processor which utilizes the method and system of the present invention;

FIG. **3** is a flow chart depicting the process of instruction execution according to one implementation of the present invention;

FIG. **4A** is a block diagram of the circuit components utilized within one embodiment of the present invention; and

FIG. **4B** is a block diagram depicting an instruction queue and register in accordance with one embodiment of the present invention.

DETAILED DESCRIPTION OF ILLUSTRATIVE EMBODIMENT

With reference now to the figures and in particular with reference to FIG. **1**, there is illustrated a block diagram of the basic structure of a data processing system **100** utilized in the preferred embodiment of the invention. Data processing system **100** has at least one central processing unit (CPU) or processor **10** which is connected to several peripheral devices, including input/output devices **114** (such as a display monitor, keyboard, and graphical pointing device) for user interface, a permanent memory device **116** (such as a hard disk) for storing the data processing system's operating system and user programs/applications, and a temporary memory device **118** (such as random access memory or RAM) that is utilized by processor **10** to implement program instructions. Processor **10** communicates with the peripheral devices by various means, including a bus **120** or a direct channel **122** (more than one bus may be provided utilizing a bus bridge).

Those skilled in the art will further appreciate that there are other components that might be utilized in conjunction with those shown in the block diagram of FIG. **1**; for example, a display adapter connected to processor **10** might be utilized to control a video display monitor, and a memory controller may be utilized as an interface between temporary memory device **118** and processor **10**. Data processing system **100** also includes firmware **124** whose primary purpose is to seek out and load an operating system from one of the peripherals (usually permanent memory device **116**) whenever the data processing system is first turned on. In the preferred embodiment, data processing system contains a relatively fast CPU or processor **10** along with sufficient temporary memory device **118** and space on permanent memory device **116**, and other required hardware components necessary for providing efficient execution of instructions.

FIG. **2** is a block diagram of processor **10**, for processing information according to a preferred embodiment of the present invention. Processor **10** may be located within data processing system **100** as depicted in FIG. **1**. In the depicted embodiment, processor **10** comprises a single integrated circuit superscalar microprocessor. Accordingly, as discussed further below, processor 10 includes various execution units, registers, buffers, memories, and other functional units, which are all formed by integrated circuitry. In a preferred embodiment of the present invention, processor **10** comprises one of the PowerPc™ line of microprocessors, which operates according to reduced instruction set computing (RISC) techniques. As depicted in FIG. 1, processor **10** is coupled to system bus **11** via a bus interface unit BIU **12** within processor **10**. BIU **12** controls the transfer of information between processor **10** and other devices coupled to system bus **11** such as a main memory (not illustrated). Processor **10**, system bus **11**, and the other devices coupled to system bus **11** together form a data processing system.

BIU **12** is connected to instruction cache **14** and data cache **16** within processor **10**. High speech caches, such as instruction cache **14** and data cache **16**, enable processor **10** to achieve relatively fast access time to a subset of data or

instructions previously transferred from main memory to instruction cache **14** and data cache **16**, thus improving the speech of operation of the data processing system. Instruction cache **14** is further coupled to sequential fetcher **17**, which fetches instructions from instruction cache **14** during each cycle for execution. Sequential fetcher **17** transmits branch instructions fetched from instruction cache **14** to branch processing unit BPU **18** for execution, but stores sequential instructions within instruction queue **19** for execution by other execution circuitry within processor **10**.

In the depicted embodiment, in addition to BPU **18**, the execution circuitry of processor **10** comprises multiple execution units, including fixed-point unit (FXU) **22**, load/store unit (LSU) **28**, and floating-point unit (FPU) **30**. As is well known by those skilled in the art, each of execution units FXU **22**, LSU **28**, and FPU **30** executes one or more instructions within a particular class of sequential instructions during each processor cycle. For example, FXU **22** performs fixed-point mathematical operations such as addition, subtraction, ANDing, ORing, and XORing utilizing source operands received from specified general purpose registers (GPRs) **32**. Following the execution of a fixed point instruction, FXU **22** outputs the data results of the instruction to GPR rename buffers **33**, which provide temporary storage for the result data until the instruction is completed by transferring the result data from GPR rename buffers **33** to one or more of GPRs **32**. Conversely, FPU **30** performs floating-point operations, such as floating-point multiplication and division, on source operands received from floating-point registers FPRs **36**. FPU **30** outputs data resulting from the execution of floating-point instructions to selected FPR rename buffers **37**, which temporarily store the result data until the instructions are completed by transferring the result data from FPR rename buffers **37** to selected FPRs **36**. As its name implies, LSU **28** executes floating-point and fixed-point instructions which either load data from memory (i.e., either data cache **16** or main memory) into selected GPRs **32** or FPRs **36** or which store data from a selected GPRs **32** or FPRs **36** to memory.

Processor **10** employs both pipelining and out-of-order execution of instructions to further improve the performance of its superscalar architecture. Accordingly, instructions can by executed by FXU **22**, LSU **28**, and FPU **30** in any order as long as data dependencies are observed. In addition, instructions are processed by each of FXU **22**, LSU **28** and FPU **30** at a sequence of pipeline stages. As is typical of high performance processors, each instruction is processed at five distinct pipeline stages, namely, fetch, decode/dispatch, execute, finish and completion.

During the fetch stage, sequential fetcher **17** retrieves one or more instructions associated with one or more memory addresses from instruction cache **14**. Sequential instructions fetched from instruction cache **14** are stored by sequential fetcher **17** within registers such as instruction queue **19**. In contrast, sequential fetcher **17** removes branch instructions from the instruction stream and forwards them to BPU **18** for execution. BPU **18** includes a branch prediction mechanism, which in one embodiment comprises a dynamic prediction mechanism such as a branch history table, that enables BPU **18** to speculatively execute unresolved conditional branch instructions by predicting whether the branch will be taken. Alternatively, in other embodiments of the present invention, a static, compiler-based prediction mechanism is implemented. As will be described in greater detail below, the present invention minimizes branch misprediction and subsequent penalties incurred by processor **10** when speculative instruction branch prediction is desired.

During the decode/dispatch stage, dispatch unit **20** decodes and dispatches one or more instructions from instruction queue **19** to the appropriate ones of execution units FXU **22**, LSU **28** and FPU **30**. Decoding involves determining the type of instruction including its characteristics and the execution unit to which it should be dispatched. In the preferred embodiment, it also involves determining whether or not the instruction is speculative. Also during the decode/dispatch stage, dispatch unit **20** allocates a rename buffer within GPR rename buffers **33** or FPR rename buffers **37** for each dispatched instructions' result data. According to a preferred embodiment of the present invention, dispatch unit **20** is connected to execution units FXU **22**, LSU **28** and FPU **30** by a set of registers described below with reference to FIGS. **4A** and **4B**.

FIG. **4A** depicts the execution units FXU **22**, LSU **28** and FPU **30** and their corresponding register **401A**, **401B**, and **401C** and queue **407A**, **407B**, and **407C**. Typically, each register **401A**, **401B**, and **401C** allocates a valid bit **403** associated with each entry/instruction queued to the execution unit FXU **22**, LSU **28**, and FPU **30** in its respective queue **407A**, **407B**, and **407C**. This bit **403** when "on" (set to "1") indicates that an instruction is present and is traditionally utilized to give prominence to that instruction within queue **401A**, **401B**, and **401C**. Threshold value **405A**, **405B**, and **405C** for execution unit FXU **22**, LSU **28**, and FPU **30** is also illustrated in FIG. **4A**. As illustrated each unit may have a unique threshold value **405A**, **405B**, and **405C**. In the preferred embodiment, the tracking of the number of instructions in each queue is implemented by counting the number of bits **403** in register **401A**, **401B**, and **401C**. Bits **401** are counted by counter **411** which is connected to register **401A**, **401B**, and **401C**. After an instruction has been executed the corresponding bit is reset (set to '0') reducing the number count for that queue in the preferred embodiment. Thus registers **401A**, **401B**, and **401C** are utilized to keep track of the number of instructions in each queue **407A**, **407B**, and **407C** of each execution unit FXU **22**, LSU **28**, and FPU **30** respectively. For the purposes of this illustrative embodiment, it is assumed that instructions are loaded sequentially into the queue of the execution units. Those skilled in the art understand that this representation does not preclude other representations and further that the instructions are not limited to being executed in the sequential manner as illustrated. It is further understood by those skilled in the art that alternative ways of tracking the number of instructions in an execution unit's queue are contemplated as falling within the scope of the present invention. Utilization of the registers as presented here is solely for illustrative purposes, and is not exclusive.

Dispatch unit **20** is also programmed to dispatch speculative instructions based on given conditions supplied to it from OR circuit **415** which is connected to comparator **413**. Comparator **413** is provided with the threshold values for each execution unit FXU **22**, LSU **28** and FPU **30**. In the preferred embodiment, when dispatch unit **20** receives a speculative branch instruction (determined during the decoding stage), it first checks the system load to determine whether or not to send the branch instruction. In the preferred embodiment, comparator **413** compares each execution unit load level (number of instructions awaiting execution) to its threshold value. The results of these comparisons are then ORed together by OR circuit **415** to yield a "false," output when all of the units are below their threshold. If any one or more unit is above its threshold, then a "true" output is yielded. In the preferred embodiment, dispatch unit **20** will only dispatch a speculative branch

instruction when the results of the OR function is false. Comparisons are made at the beginning of each cycle, while a speculative branch instruction is present. When the result is true, the speculative branch instruction is held until a false result is obtained. A false result may thus be obtained in the next cycle or in subsequent cycles. Those skilled in the art are familiar with the functioning of an OR circuit.

FIG. **4**A illustrates one embodiment of how the checking stage of the invention is implemented. Threshold value **405**A, **405**B, and **405**C for each unit FXU **22**, LSU **28** and FPU **30**, is represented on its corresponding queue **407**A, **407**B and **407**C. The number of instructions in a given queue is determined by checking bits **403** of register **401**A, **401**B, and **401**C. In the illustrative embodiment, FXU **22** has less instructions in its queue **407**A than its threshold value **405**A. Load/Store Unit **28** also has less instructions in its queue **407**B than its threshold value **405**B. A comparison by comparator **413** yields a false result for both units. FPU **30** has more instructions than its threshold value **405**C and yields a true result when a comparison is done by comparator **413**. ORing these results together thus yields a true result and the speculative branch instruction is not dispatched.

In another embodiment of the invention, only the number of speculative instructions found within any one queue is tracked. This yields a more realistic picture of how speculative the branch instruction may be. Further, the threshold value is set as a predetermined number of such speculative instructions found in the queue. When a speculative branch instruction is presented, the comparison is made based on these two values. Thus in FIG. **4**B, although FPU **30** has more total instructions **409** than the threshold value, only the instructions which are themselves speculative are counted. Instruction **1** **409**A and Instruction **3** **409**B are speculative. Comparator **413** checks this number of speculative instructions (2) against threshold value **405**C. Identifying speculative instructions in this embodiment entails keeping track of the instruction when it is dispatched from dispatch unit. In this embodiment, register **401**C is provided with an additional bit **404** for tracking whether or not an instruction is speculative.

The preferred embodiment of the invention is implemented on a general system level and determines the dispatching of speculative branches. Another embodiment of the invention is implemented on an individual execution unit level and determines the dispatching of a speculative instruction which targets a specific execution unit. Dispatch unit **20** checks the target execution unit's register to determine the number of instructions present in the unit's queue. Comparator **413** then compares this value with the corresponding threshold value. As in the preferred embodiment of the invention, when the threshold value is equal to or below the number of instructions in the unit's queue, dispatch unit **20** holds the speculative instruction until the next cycle or subsequent cycles. If the number of instructions in the unit's queue falls below the threshold value, dispatch unit **20** dispatches the speculative instruction to the execution units to await execution.

Those skilled in the art understand that although specific methods have been disclosed of determining the threshold value of an execution unit, the threshold value is in fact an arbitrary value which may be selected in a variety of ways. The invention as described contemplates all such selection methods.

During the execution stage, execution units FXU **22**, LSU **28** and FPU **30** execute instructions received from dispatch unit **20** as soon as the source operands for the indicated operations are available. After execution has terminated, execution units FXU **22**, LSU **28**, and FPU **30** store data results within either GPR rename buffers **33** or FPR rename buffers **37**, depending upon the instruction type. Then, execution units FXU **22**, LSU **28**, and FPU **30** signal completion unit **40** that the execution unit has finished an instruction. Finally, instructions are completed in program order by transferring result data from GPR rename buffers **33** or FPR rename buffers **37** to GPRs **32** or FPRs **36**, respectively.

The execution of instructions prior to the final possible definition of all conditions effecting execution is called speculative execution. To wait for the outcome of conditional branches, or the arrival of all possible interrupts, would make full concurrent processing impossible.

The present invention provides a novel method to disable speculative execution when resources are busy. In the preferred embodiment, this is achieved by allowing branch guessing only when it is likely that processor units will benefit from the speculation. More specifically, in the preferred embodiment, the dispatch of a speculative branch path is dependant on a measure of current utilization of the execution units. In general, if there are instructions executing and instructions queued up to be executed, then the need for additional instructions to be dispatched is less than if the execution units were waiting for work (i.e., new instructions to execute). Therefore, in the preferred embodiment of this invention, a decision to dispatch speculative instructions is based on the current utilization of the execution units. In one illustrative embodiment of the invention, the logic tracks how many instructions are queued up for each of the execution units and, based on a threshold for each execution unit, decides whether or not to dispatch speculative paths.

In the preferred embodiment of the invention, different units are assigned different threshold settings. Threshold levels are based on the unit's individual characteristics, such as, the type of instructions executed on the unit. For the purposes of this invention, a unit's threshold refers specifically to a number of instructions which are located within the unit's instruction queue awaiting execution. This number is variable depending on the system loading and the developer's choice. In the preferred embodiment, the threshold is typically a number less than the maximum number of instructions which can be stored in the unit's instruction queue. For instance, in the case of instructions which could be executed in Memory Load/Store execution units, the latency of these instructions may be large due to the nature of memory and cache management, and therefore, the threshold level for the Load/Store units is lower than for a unit that didn't have a memory latency involved in its operation.

In another embodiment of the invention, the number of instructions which had been dispatched (but not yet completed) is tracked as an indication of current execution unit utilization or as an indication of the amount of work (i.e., instructions) that was already assigned to the execution unit.

In one embodiment, the invention is implemented without pre-set threshold levels. The instructions are fetched and examined in the predicted path to identify which execution unit(s) would be the target for execution of these instructions and then, based on how many instructions were queued up (to these units). When the target unit had instructions queued up, then speculative instructions wouldn't be dispatched to this unit. When the unit is idle, however, speculative instructions would be dispatched to it. Those skilled in the art

understand that the setting of a threshold value, though disclosed as the preferred embodiment, is not essential to the working of the present invention when implemented as described above.

The invention is preferably implemented in hardware and may be illustrated in terms of internal circuitry of a processor for enabling speculative branch dispatching as in FIG. 4A. In this embodiment, counter **411** tracks the number of instructions waiting to be executed in each execution unit. Comparator **413** indicates when the number of instructions waiting to be executed exceeds a predetermined threshold. Also, OR circuit **415** then compares a plurality of comparison results to determine system loading. These circuit components are additional components coupled to each other and the existing components of a processor, namely dispatch unit **20**, instruction queue **407**, and register **401** to enable speculative dispatching only when a number of instructions waiting for execution is less than a is predetermined threshold.

The above embodiment consists of new circuitry components. Another embodiment of the invention utilizes current hardware components. The dispatching circuitry for example, currently has a large list of conditions which are checked prior to dispatching of an instruction. Adding additional conditions to implement the invention requires very little effort and may be preferred in certain circumstances. Additionally, the physical components described above, namely a comparator, a counter and an OR circuit may be implemented as software blocks within the data processing system. The invention is capable of being implemented in any system/processor. In the preferred embodiment, a superscalar processor with multiple execution units capable of handling multiple numbers of instruction simultaneously, is desired.

It is understood by those skilled in the art that instructions may exist in more than one state. An instruction may be waiting to be executed or it may have already been executed and is in some stage of completeness. The present invention contemplates setting threshold values and counting the number of instructions in the queue based on predetermined factors which the developer may implement. For instance, if an instruction will take a large number of cycles before it is completed, then it would be preferred to count that instruction. If, however, that same instruction has been executed and will be completed in the current or next cycle, then it would be preferred to not count the instruction.

FIG. **3** depicts a flow chart of the process involved in the implementation of the preferred embodiment of the present invention. The process begins (step **201**) with the system developer setting threshold values for each execution unit (step **203**). During running of an application, an instruction is fetched (step **205**) by the sequential fetcher and placed into a register (step **207**). The system then checks to see if the instruction is speculative, i.e. if it is a branch instruction and if branch guessing is required (step **209**). When the instruction is speculative, a further check is made to determine if the execution unit queue is at its threshold level (step **210**). As discussed above, there are several ways to make this determination. For example, a register attached to each execution unit is utilized to track the number of instructions queued and this number is compared against the threshold value by a comparator. If the queue is at its threshold value, the instruction is held in abeyance until some of the prior instructions in the queues are executed (step **211**). This may take several cycles; however, the instruction is presented again to determine whether or not it is still speculative (step **209**). If, however, the instruction queue is not at its threshold level, or when the instruction is not speculative, then it is sent to the execution unit's instruction queue (step **213**) and ultimately executed (step **215**). The process then checks memory for additional instructions to execute (step **217**). If more instructions are available then the process returns to fetch the next instruction (step **205**). Otherwise, the process ends (step **219**). It is understood by those skilled in the art that although the above example has been shown with reference to single individual instructions, any number of instructions may be fetched from memory and processed simultaneously depending on hardware capabilities.

While the invention has been particularly shown and described with reference to a preferred embodiment, it will be understood by those skilled in the art that various changes in form and detail may be made therein without departing from the spirit and scope of the invention.

What is claimed is:

1. A method for dispatching of instructions in a data processing system comprising the steps of:

determining a current utilization measure for one or more execution units in said data processing system, wherein said current utilization measure is a number of instructions queued at said one or more execution units;

setting a threshold value for each execution unit, wherein an individual threshold is established for each execution unit, wherein said threshold value is selected based on characteristics of each of said plurality of execution units, and said threshold value is a number corresponding to the maximum desired number of instructions queued at said execution unit above which no speculative dispatch occurs;

comparing said number of instructions queued at each of said execution units with said threshold value of each of said execution units, respectively, to produce a series of results, and ORing said series of results, wherein said ORing yields a false output when no execution unit is at its threshold, and a true output when any one of said plurality of execution units is at its threshold; and

dispatching a speculative branch instruction to a selected execution unit only when said ORing step yields said false output indicating that said current utilization measure is less than said predetermined threshold value.

2. The method of claim **1**, wherein said determining step is implemented by tracking said number of instructions which have been dispatched, but not yet completed, as an indication of current execution utilization.

3. The method of claim **1**, wherein said determining step is implemented by tracking said number of instructions which have been dispatched, but not yet executed, as an indication of the amount of work that has already been assigned to said execution unit.

4. The method of claim **1**, wherein said dispatching step further includes:

first fetching an instruction;

examining said instruction in a predicted path to identify which execution unit is a target for execution of said instruction;

examining said instruction unit to determine the number of instructions queued up to said unit;

when said instruction unit has instructions queued up, disabling speculative instruction dispatching to said unit; and

when said instruction unit is idle, enabling the dispatch of speculative instructions to said unit.

5. A data processing system for dispatching of instructions comprising:

means for determining a current utilization measure for one or more execution units in said data processing system determining a current utilization measure for one or more execution units in said data processing system, wherein said current utilization measure is a number of instructions queued at said one or more execution units;

means for setting a threshold value for each execution unit, wherein an individual threshold is established for each execution unit, wherein said threshold value is selected based on characteristics of each of said plurality of execution units, and said threshold value is a number corresponding to the maximum desired number of instructions queued at said execution unit above which no speculative dispatch occurs;

means for comparing said number of instructions queued at each of said execution units with said threshold value of each of said execution units, respectively, to produce a series of results, and ORing said series of results, wherein said ORing yields a false output when no execution unit is at its threshold, and a true output when any one of said plurality of execution units is at its threshold; and

means for dispatching a speculative branch instruction to a selected execution unit only when said ORing step yields said false output indicating that said current utilization measure is less than said predetermined threshold value.

6. The data processing system of claim 5, wherein said determining means includes means for tracking said number of instructions which have been dispatched, but not yet completed, as an indication of current execution utilization.

7. The data processing system of claim 5, wherein said determining means includes means for tracking said number of instructions which have been dispatched, but not yet executed, as an indication of the amount of work that has already been assigned to said execution unit.

8. The data processing system of claim 5, wherein said dispatching means further includes:

means for first fetching an instruction;

means for examining said instruction in a predicted path to identify which execution unit is a target for execution of said instruction;

means for examining said instruction unit to determine the number of instructions queued up to said unit;

when said instruction unit has instructions queued up, means for disabling speculative instruction dispatching to said unit; and

when said instruction unit is idle, means for enabling the dispatch of speculative instructions to said unit.

9. The data processing system of claim 5, wherein:

said determining means includes a register coupled to an instruction queue of said execution unit, said instruction queue holding instructions awaiting execution by said execution unit, and said register storing said predetermined threshold value and tracking the number of instructions in said instruction queue;

said comparing means includes a comparator connected to said register via a counter, said counter for counting the number of instructions in said instruction queue, and comparator for comparing said number of instructions with said predetermined threshold value to yield a result, and further wherein when more than one execution unit is checked, said comparing means further includes a comparator for ORing said result of a first comparison with said result of a second comparison; and

said dispatching means includes a dispatch unit coupled to said comparator and said instruction queue, wherein said dispatch unit receives instructions form said instruction fetcher, and when said instruction is speculative, said dispatch unit dispatches said instruction to said instruction queue of said execution unit only when said comparator yields a "false" output to said dispatch unit.

10. A data processor comprising:

one or more execution units, wherein an execution unit has an instruction queue for holding instructions awaiting execution in said execution unit;

a register coupled to said instruction queue of said execution unit, wherein said register stores a predetermined threshold value for said instruction queue and further wherein said register tracks a number of instructions in said instruction queue;

a counter connected to said register, said counter for counting a number of instructions in said instruction queue;

a comparator connected to said counter for comparing said number of instructions in said instruction queue with said predetermined threshold value to yield a result, wherein said comparator performs a comparison on more than one execution unit to yield a plurality of results, wherein further said comparator is coupled to an OR circuit which performs an ORing function on said plurality of results to yield a second result; and

a dispatch unit coupled to an instruction fetcher, said comparator and said instruction queue, wherein said dispatch unit receives instructions from said instruction fetcher, and when said instruction is speculative, said dispatch unit dispatches said instruction to said instruction queue of said execution unit only when said comparator yields a particular result.

11. The data processor of claim 10, wherein said second result is either said particular result or a different result, and wherein said second result is provided to said comparator.

12. The data processor of claim 10, wherein further said register stores said predetermined threshold value.

* * * * *

US005506967A

United States Patent [19]

Barajas et al.

[11] Patent Number: 5,506,967

[45] Date of Patent: Apr. 9, 1996

[54] **STORAGE QUEUE WITH ADJUSTABLE LEVEL THRESHOLDS FOR CACHE INVALIDATION SYSTEMS IN CACHE ORIENTED COMPUTER ARCHITECTURES**

[75] Inventors: **Saul Barajas**, Mission Viejo; **David M. Kalish**, Laguna Niguel; **Bruce E. Whittaker**, Mission Viejo, all of Calif.

[73] Assignee: **Unisys Corporation**, Blue Bell, Pa.

[21] Appl. No.: **78,361**

[22] Filed: **Jun. 15, 1993**

[51] **Int. Cl.**[6] **G06F 13/00**

[52] **U.S. Cl.** ..... **395/250**; 395/440; 364/DIG. 1; 364/DIG. II

[58] **Field of Search** 395/250, 425, 395/400; 364/243.41, 244.3

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,195,340	3/1980	Joyce	395/425
4,663,706	5/1987	Allen et al.	395/200
4,829,475	5/1989	Ward et al.	365/78
4,864,543	9/1989	Ward et al.	365/221
5,025,365	6/1991	Mathur et al.	364/243.4
5,058,006	10/1991	Durdan et al.	364/228.1
5,084,841	6/1992	Williams et al.	365/189.07
5,113,418	5/1992	Szezepanek et al.	375/118
5,157,774	10/1992	Culley	395/425
5,193,163	3/1993	Sanders et al.	395/425
5,218,670	6/1993	Sodek, Jr. et al.	395/115
5,265,233	11/1993	Frailons et al.	395/425
5,317,720	5/1994	Stamm et al.	395/425

Primary Examiner—Parshotam S. Lall
Assistant Examiner—Zarni Maung
Attorney, Agent, or Firm—Alfred W. Kozak; Mark T. Starr; Robert R. Axenfeld

[57] **ABSTRACT**

In a time-shared bus computer system with processors having cache memories, an adjustable invalidation queue for use in the cache memories. The invalidation queue has adjustable upper and lower limit positions that define when the queue is logically full and logically empty, respectively. The queue is flushed down to the lower limit when the contents of the queue attain the upper limit. During the queue flushing operation, WRITE requests on the bus are RETRYed. The computer maintenance system sets the upper and lower limits at system initialization time to optimize system performance under maximum bus traffic conditions.

10 Claims, 4 Drawing Sheets

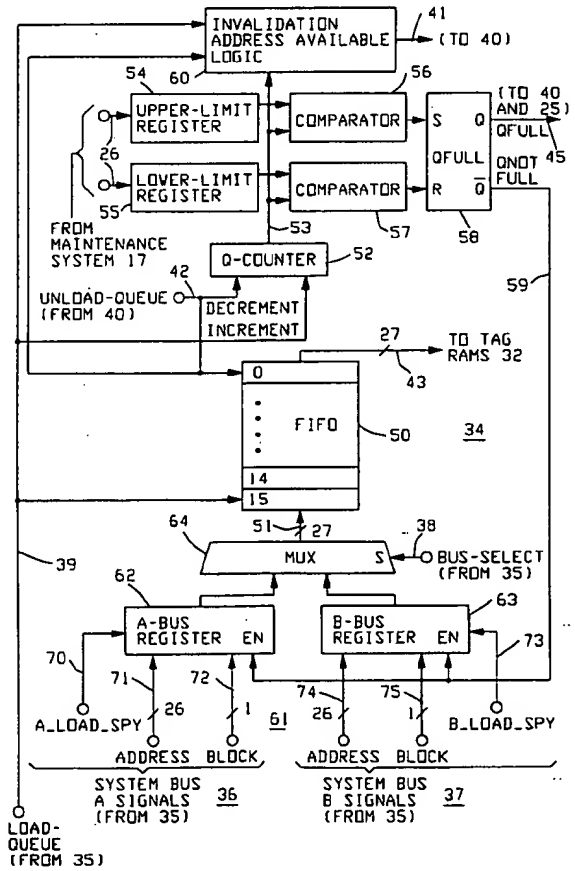
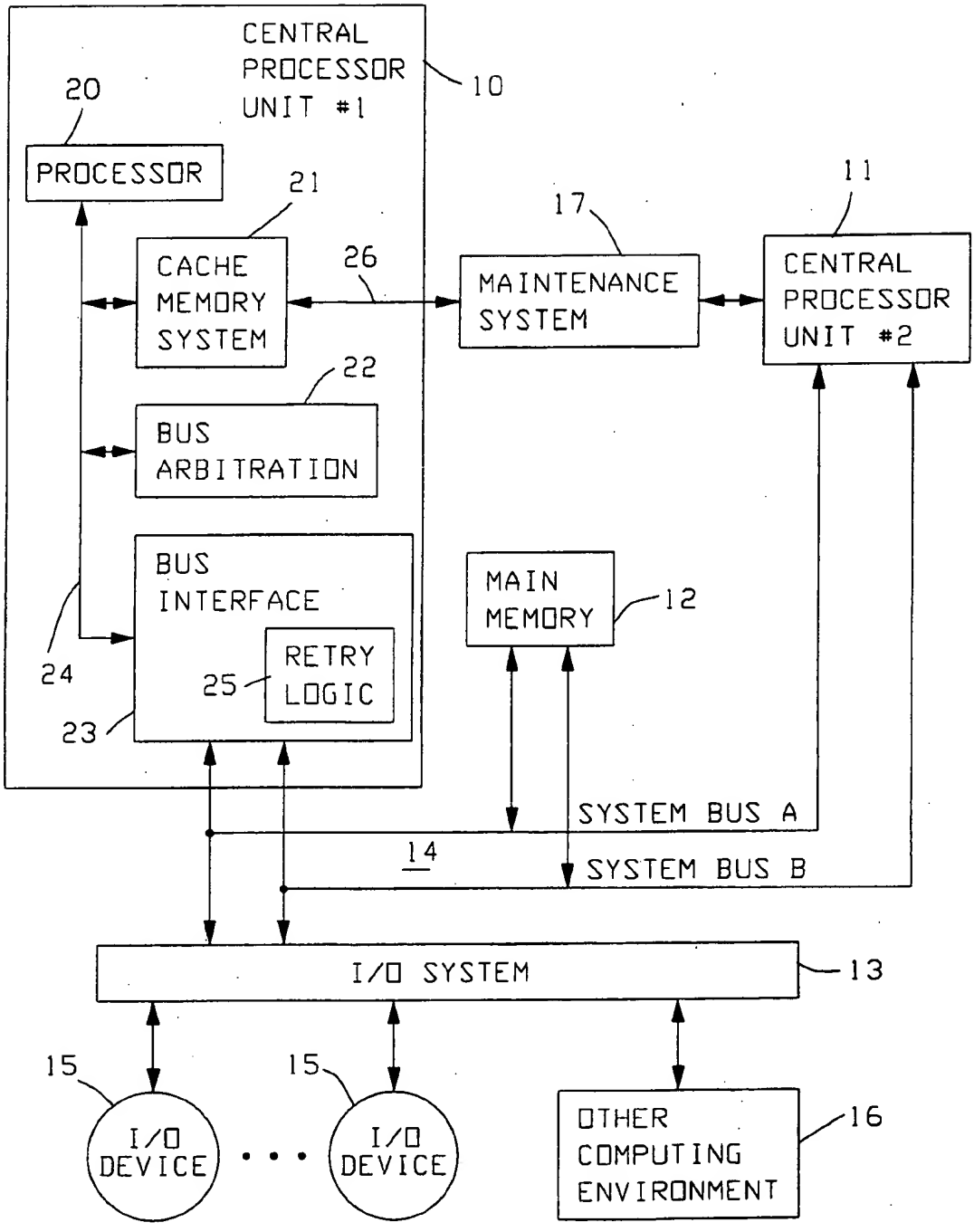


FIG. 1


CENTRAL PROCESSOR UNIT #1
10
20
PROCESSOR
21
CACHE MEMORY SYSTEM
26
17
MAINTENANCE SYSTEM
11
CENTRAL PROCESSOR UNIT #2
22
BUS ARBITRATION
BUS INTERFACE
RETRY LOGIC
25
24
23
MAIN MEMORY
12
SYSTEM BUS A
14
SYSTEM BUS B
I/O SYSTEM
13
15
15
I/O DEVICE
I/O DEVICE
OTHER COMPUTING ENVIRONMENT
16

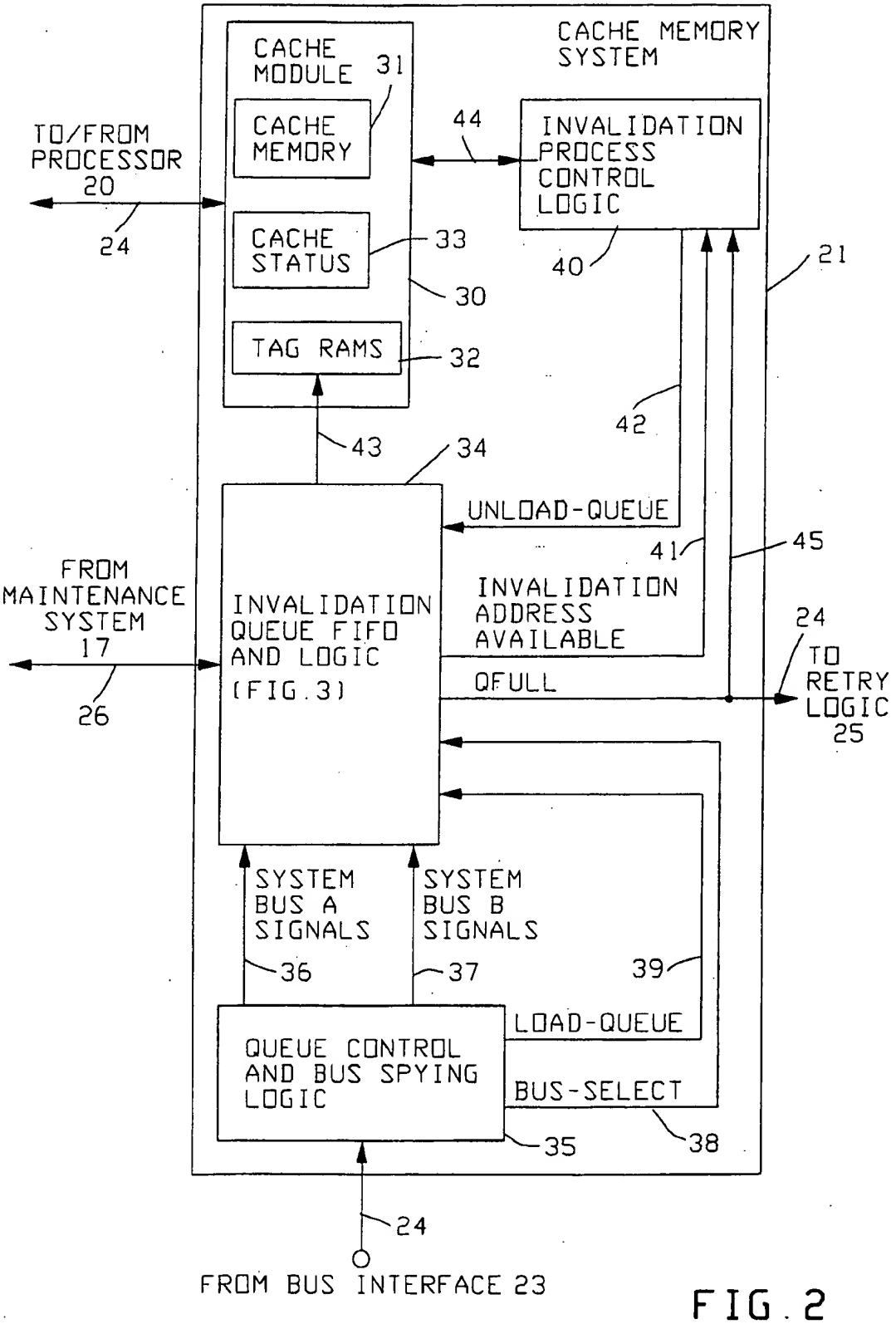

CACHE MEMORY SYSTEM
CACHE MODULE
31
CACHE MEMORY
CACHE STATUS
33
30
TAG RAMS
32
TO/FROM PROCESSOR 20
24
44
INVALIDATION PROCESS CONTROL LOGIC
40
21
42
43
34
UNLOAD-QUEUE
41
45
FROM MAINTENANCE SYSTEM 17
26
INVALIDATION QUEUE FIFO AND LOGIC (FIG. 3)
INVALIDATION ADDRESS AVAILABLE
24
TO RETRY LOGIC 25
QFULL
SYSTEM BUS A SIGNALS
SYSTEM BUS B SIGNALS
36
37
39
LOAD-QUEUE
QUEUE CONTROL AND BUS SPYING LOGIC
BUS-SELECT
35
38
24
FROM BUS INTERFACE 23

FIG. 2

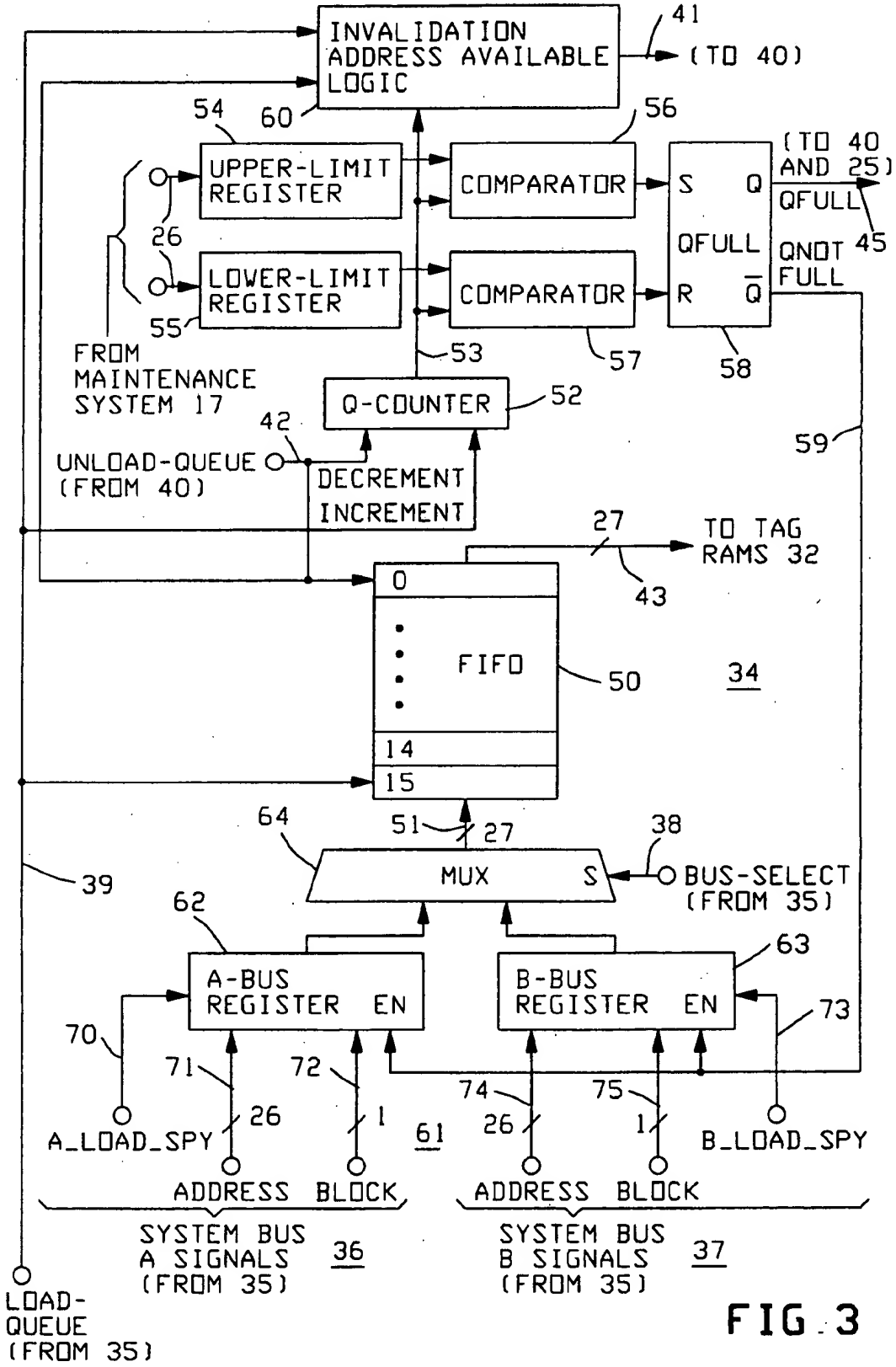

INVALIDATION ADDRESS AVAILABLE LOGIC
41
(TO 40)
54
60
56
UPPER-LIMIT REGISTER
COMPARATOR
S Q
QFULL
R Q̄
(TO 40 AND 25)
QFULL
QNOT FULL
45
26
LOWER-LIMIT REGISTER
COMPARATOR
55
53
57
58
FROM MAINTENANCE SYSTEM 17
Q-COUNTER
52
42
59
UNLOAD-QUEUE (FROM 40)
DECREMENT
INCREMENT
27
TO TAG RAMS 32
43
0
FIFO
50
34
14
15
64
51
27
38
MUX
S
BUS-SELECT (FROM 35)
39
62
63
A-BUS REGISTER EN
B-BUS REGISTER EN
73
70
71
72
74
75
26
1
26
1
A_LOAD_SPY
61
B_LOAD_SPY
ADDRESS BLOCK
ADDRESS BLOCK
SYSTEM BUS A SIGNALS 36 (FROM 35)
SYSTEM BUS B SIGNALS 37 (FROM 35)
LOAD-QUEUE (FROM 35)

FIG. 3

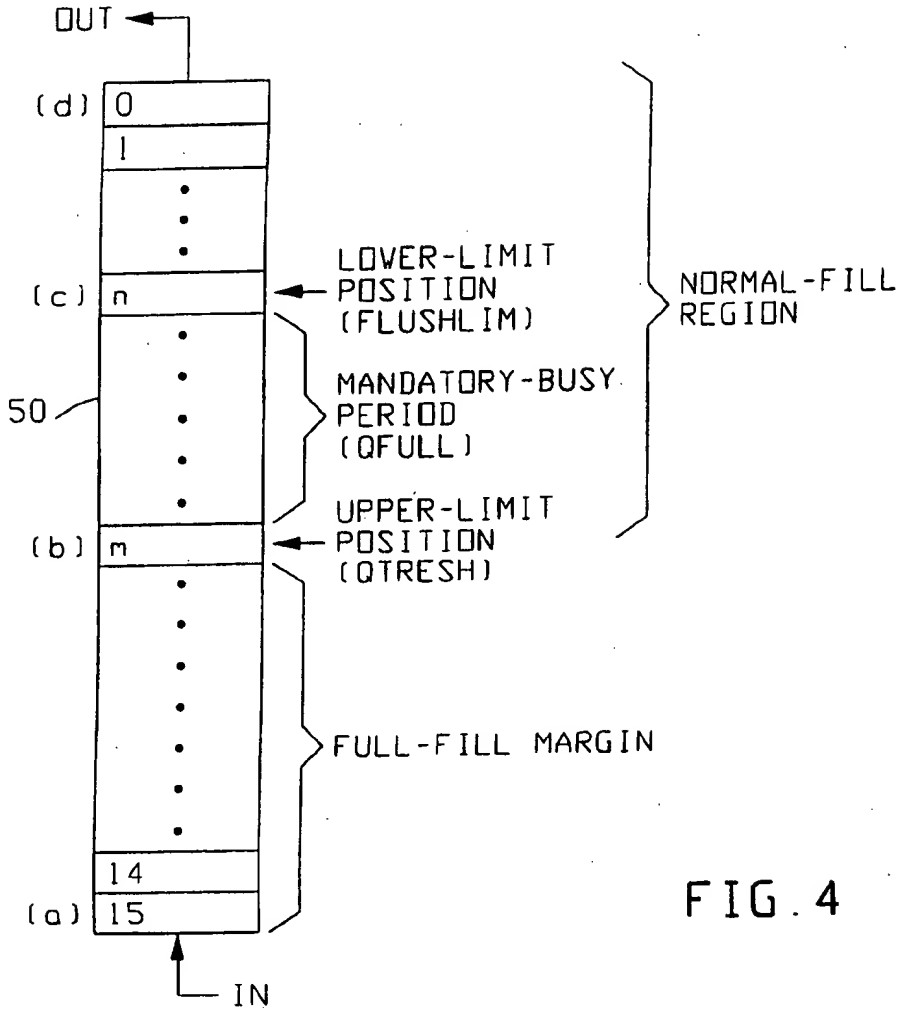
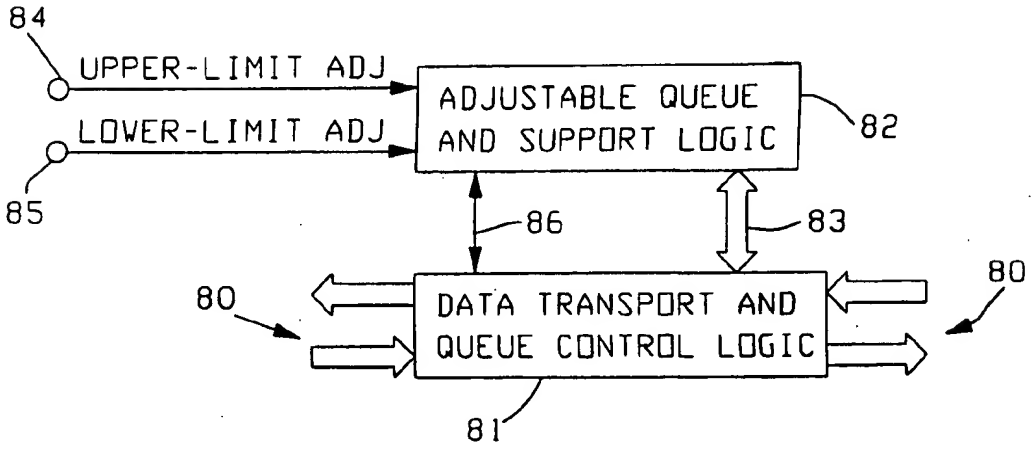

OUT
(d)
0
1
(c)
n
LOWER-LIMIT POSITION (FLUSHLIM)
NORMAL-FILL REGION
50
MANDATORY-BUSY PERIOD (QFULL)
UPPER-LIMIT POSITION (QTRESH)
(b)
m
FULL-FILL MARGIN
14
(a)
15
IN

FIG. 4

FIG. 5


84
UPPER-LIMIT ADJ
LOWER-LIMIT ADJ
85
ADJUSTABLE QUEUE AND SUPPORT LOGIC
82
86
83
80
80
DATA TRANSPORT AND QUEUE CONTROL LOGIC
81

STORAGE QUEUE WITH ADJUSTABLE LEVEL THRESHOLDS FOR CACHE INVALIDATION SYSTEMS IN CACHE ORIENTED COMPUTER ARCHITECTURES

BACKGROUND OF THE INVENTION

1. Field of the Invention

The invention relates to buffer queue structures particularly with respect to utilization in the cache invalidation logic of a cache oriented computer architecture.

2. Description of the Prior Art

Present day digital computer architectures often include interconnected subsystems comprising a plurality of central processor modules, a main memory subsystem and one or more Input/Output (I/O) subsystems. The central processor modules, main memory and I/O subsystems preferably intercommunicate by a time-shared bus system intercoupling the component sections of the computer system. In this architecture, each central processor module may include a private cache into which the processor copies words from main memory utilizing the cache in performing its processes. For example, a processor may copy program instructions and data from main memory to its cache and, thereafter, execute the program task from cache. As is appreciated, cache is used in this manner to enhance performance. The cache memory is significantly faster than main memory and the processor with the cache avoids going back and forth on the bus to main memory for each instruction.

A problem recognized in such systems is that of cache consistency. The data in the cache memories and main memory must be maintained coherent and updated with respect to each other. All copies of information at a specific address in all of the memory facilities must be maintained identical. For example, if a first one of the processors executes a WRITE TO MEMORY overwriting a main memory location that had been copied by a second one of the processors into its cache, the data in that location of the cache of the second processor becomes obsolete and invalid. The cache inconsistency condition is exacerbated when, for example, an I/O subsystem streams data into main memory overwriting numerous memory locations cached by the processors.

Traditionally, computer systems with cache memories maintain data integrity by using a cache invalidation process. The process involves each cache system monitoring, or spying upon, the memory operations of the other processors and subsystems in the computer. This is conveniently accomplished by monitoring the memory write operations on the bus. When a memory write operation is detected, each cache memory system must, at some time, execute an internal cache invalidation operation or cycle. The cache invalidation cycle involves testing the contents of the cache for the specific address of the write operation that was detected. If the cache memory system determines that it contains this address, the system marks the address as invalid. The processor with the cache must update the contents of an invalid cache location before using it.

The execution time of a cache invalidation cycle occupies a significant number of bus cycles. Thus, while the invalidation process is executing, the associated processor is prevented from performing program tasks since its cache memory resources, such as the cache tag RAM accessing and invalidation resources, are busy with the invalidation process. Additionally, when the cache memory resources of

a processor are occupied with a cache invalidation cycle, other processors may perform WRITES TO MEMORY that will not be detected by the busy cache system. Such an occurrence is catastrophic to the computer system since data coherency is destroyed.

In a prior computer design, when a cache invalidation cycle is in progress, the cache system puts a RETRY signal on the bus in response to a write request from another processor or subsystem, advising the other processor or subsystem to retry its memory write request at a later time. Under conditions of heavy bus traffic, the RETRY mechanism approach may prevent processors from achieving adequate bus access thereby preventing the useful work thereof. A processor may be excessively RETRYed degrading its performance. By excessively RETRYing the I/O subsystem, this approach may also seriously interfere with the I/O subsystem bus traffic thereby detrimentally impeding the entry of data into the computer system. The condition is particularly severe in an I/O subsystem burst mode where data is streamed into main memory.

Other problems of such systems and solutions thereof are described in co-pending U.S. patent application Ser. No. 08/003,352; filed Jan. 12, 1993; entitled "Inhibit Write Apparatus And Method For Preventing Bus Lockout"; and U.S. patent application Ser. No. 08/016,588; filed Jan. 11, 1993; entitled "Varying Wait Interval Retry Apparatus And Method For Preventing Bus Lockout"; both applications by T. C. White et al. and assigned to the Assignee of the present invention. Said Ser. Nos. 08/003,352 and 08/016,588 are incorporated herein by reference.

Systems of the type described may also include an invalidation buffer or queue associated with each cache system for buffering the information detected by the spying system required for specific invalidations. The invalidation process control system may then execute the process at the least inconvenient time for the associated processor. When the invalidation queue becomes full, the RETRY signal may be issued to the bus in response to attempted WRITES with the concomitant disadvantages discussed above.

The invalidation queue should be deep enough to hold sufficient writes to minimize the effect of the cache invalidation cycles on the processor performance while providing that no WRITES are lost. The queue will fill up too quickly if the depth thereof is too shallow whereby insufficient writes are held. When the queue becomes full, the invalidation process executions are mandatory and the work of the associated processor must terminate until the invalidation processing is completed and the queue is no longer full. This is denoted as a queue flushing process. Additionally, when the queue becomes full, there is an increased possibility of losing a write on the system bus. Such a situation is catastrophic to the computer system since data coherency is destroyed.

As discussed above, another consequence of the queue being full is that whenever a new write operation is detected on the system bus, it is RETRYed since there is no room in the queue. The source module of the WRITE operation is then forced to repeat the entire operation, at which time the queue may or may not have room for the new invalidation. The RETRY mechanism can significantly impede data flow on the system bus and can be so detrimental that no useful work is performed. Under such conditions, the performance of the I/O system can be seriously degraded having a detrimental effect on the entire computer system because of the impeded I/O bus traffic. The performance of the processor with the full queue is also seriously diminished as discussed above.

Conversely, it is also undesirable to configure the queue too deep since queues are expensive structures in both cost and hardware area. Additionally, the deeper the queue, the more extensive and complicated is the control logic for supporting the queue.

The depth of the queue should be an optimum size for the relative logical speeds of the incoming system bus write operations and the outgoing invalidation processes using the cache tag RAM accessing and invalidation resources. The Application Specific Integrated Circuit Very Large Scale Integration (ASIC VLSI) gate array type technologies are advantageously utilized in constructing a highly efficient invalidation queue structure. The optimum queue depth is preferably empirically determined when actually running the cache in an operating computer system utilizing the system bus or buses under maximum system conditions. However, when this can be accomplished, the ASIC device has already been designed and constructed and cannot readily be modified in any practical, rapid or cost-effective manner and without significant schedule delays. Even if optimum queue depth is achievable, should the queue become full, invalidation WRITES could be lost with the catastrophic effects discussed above. An additional disadvantage of the system described occurs because the system bus is RETRYed during the queue flushing operation. In the time required to flush a deep queue to zero, the bus can be excessively RETRYed with the concomitant disadvantages discussed above.

SUMMARY OF THE INVENTION

The above disadvantages of the prior art are obviated by an invalidation queue structure with adjustable upper and lower level limits. The structure is utilized in a computer system having first and second memory systems where the second memory system is a cache memory for storing data resident in the first memory system. In operation of the computer system, addressable locations of the first memory are overwritten thereby potentially invalidating locations in cache. A spy system monitors when addressable locations of the first memory are overwritten and provides invalidation address signals representative of the overwritten locations. The queue structure stores the invalidation address signals and an invalidation system in the cache memory system withdraws the queued address signals from the queue structure marking locations of the cache memory system invalid in accordance therewith. An upper limit determining means provides a QFULL signal when the number of invalidation address signals in the queue structure reaches an adjustable upper limit and a lower limit determining means provides a QNOTFULL signal when the number of invalidation address signals in the queue structure reaches an adjustable lower limit. The invalidation system flushes the queue structure in response to the QFULL signal and discontinues the flushing operation in response to the QNOTFULL signal. Means are included for setting the adjustable upper and lower limits in accordance with the empirical operation of the computer system to optimize system performance.

The invention also includes setting the upper limit at less than the maximum capacity of the queue so as to receive and store invalidation addresses that would otherwise be lost if the upper limit were set at maximum queue capacity.

The invention further encompasses setting the lower limit sufficiently high to reduce queue flushing time so as to minimize bus RETRY, and to minimize processor waiting times.

The adjustable queue structure also has utility in other environments such as data communication systems.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a schematic block diagram illustrating a cache oriented, time-shared bus computer architecture in which the present invention can advantageously be utilized.

FIG. **2** is a schematic block diagram illustrating the cache system of FIG. **1** in which the present invention is utilized.

FIG. **3** is a schematic block diagram illustrating details of the invalidation queue structure of FIG. **2**.

FIG. **4** is a schematic diagram of the queue structure of FIG. **3** illustrating queue adjustment parameters.

FIG. **5** is a schematic block diagram illustrating utilizing the adjustable queue of the present invention in a bi-directional data communication system.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

Referring to FIG. **1**, a computer system is illustrated comprising first and second central processor units **10** and **11**, a main memory system **12** and an I/O system **13** interconnected by a dual bus system **14**. The central processor units **10** and **11** are identical with respect to each other and may, for example, comprise microprocessors for concurrent execution of programs. The main memory system **12** stores the user software instructions and data for the operations to be performed by the computer system. The I/O system **13** couples I/O peripherals **15** into the system which may include disk, tape, printer, and the like. Other cooperative computing environments may couple to the computer system of FIG. **1** through the I/O system **13**, as schematically illustrated at **16**. The bus system **14** is a conventional time-shared dual bus denoted as system bus A and system bus B, each having data lines to transmit data, request lines for the modules **10**, **11** and **13** to acquire the bus and a grant line granting a requesting module access to the bus. The computer system of FIG. **1** also includes a conventional maintenance system **17** illustrated connected to the processing units **10** and **11**. The maintenance system **17** performs conventional functions and, in addition, provides parameters used by the present invention.

The central processor unit **10** includes a processor **20**, a cache memory system **21**, bus arbitration logic **22** and a bus interface **23** interconnected by an internal central processor unit bus **24**. The bus interface **23** includes RETRY logic **25** for RETRYing WRITE requests on the bus system **14**.

The processor **20** and the corresponding processor in the central processor unit **11** may comprise respective microprocessors for concurrent execution of programs. The cache system **21** includes a fast memory for storing instructions and data read from the main memory **12** by the processor **20**, so that the processor **20** can operate thereupon without requiring numerous accesses to the bus system **14** and the main memory **12**. As is appreciated, a significant enhancement in performance is achieved by this arrangement. The cache system **21** receives an input from the maintenance system **17** via a bus **26** for reasons to be later described.

The bus arbitration logic **22** together with the corresponding bus arbitration logic in the central processor unit **11**, in response to bus requests from the processor **20** and the corresponding processor in the central processor unit **11**, controls the bus grants and implements a priority system for resolving bus contention when two or more units simulta-

neously request access to the bus. The bus arbitration logic **22** and the corresponding logic in the central processor unit **11** preferably utilize a priority assignment protocol that tends to grant equal bus access to all bus requesters. The components **20–22** and **25** of the central processor unit **10** intercommunicate with the main bus system **14** via the bus interface **23**. The RETRY logic **25** retries WRITE requests on the bus system **14** in response to a QFULL signal from the cache memory system **21** indicating that the invalidation queue thereof is logically full, in a manner to be later described.

As discussed above, data coherence must be maintained among copies of the same data in the cache memory system **21**, the corresponding cache memory system in the central processor unit **11** and in the main memory **12**. For example, invalid data in the cache **21** would result if the I/O system **13** would overwrite locations of the main memory **12** from which the processor **20** had cached data in the cache system **21**. As discussed above, the cache system **21** includes a spy system for monitoring the system buses **14** for WRITES to the other memories, such as the main memory **12**, of the computer system of FIG. **1**. The cache system **21** further includes an invalidation system to determine if the detected WRITE addresses are in the cache and to mark these cache locations invalid. The invalidation system of the cache system **21** operates in accordance with the present invention, in a manner to be described, so as to minimize interfering in the performance of the processor **20** and in the operation of the computer system of FIG. **1**, particularly with respect to the operation of the I/O system **13**.

Referring to FIG. **2**, in which like reference numerals indicate like components with respect to FIG. **1**, details of the cache memory system **21** are illustrated. The cache memory system **21** includes a conventional cache module **30** comprising a cache data memory **31**, cache tag RAMs **32**, cache status indicators **33** and associated logic (not shown). The cache memory **31** holds the data from the main memory **12** cached therein by the processor **20**. The cache tag RAMs **32** hold the address tag and validity data for the locations of the memory **31** in a well-known manner. When the processor **20** requires access to an addressable location, the tag RAMs **32** are consulted to determine if the address is resident in the memory **31**. The cache status indicators **33** indicate the status of the tag RAMs **32** as either busy or available. The processor **20** utilizes the cache status indicators **33** to determine if the cache module **30** is available to it for program processing or is otherwise busy with an invalidation cycle. In a similar manner, the invalidation apparatus of the cache memory system **21** utilizes the cache status indicators **33** to determine if the cache module **30** is available for invalidation cycles or is busy with the processor **20**.

The cache memory system **21** includes an invalidation queue First-In First-Out (FIFO) buffer **34** and associated logic. The invalidation queue **34** holds potential invalidation addresses spied from the system buses **14** for entry into the invalidation fields of the tag RAMs **32**. The invalidation queue FIFO **34** includes adjustable upper and lower level limits for enhanced performance in accordance with the present invention. Details of the invalidation queue FIFO **34** and the associated logic will be later described with respect to FIG. **3**.

The cache memory system **21** further includes queue control and bus spying logic **35**. The queue control and bus spying logic **35** accesses the system buses **14** through the bus interface **23**. The logic **35** decodes the bus commands to detect when WRITE operations are performed and loads the appropriate address signals for such commands into spy registers within the invalidation queue FIFO and logic **34**. The system bus A address signals are applied via a path **36** and the system bus B address signals are applied via a path **37**. The logic **35** provides a binary bus-select signal to the logic **34** via a line **38** indicating whether the invalidation address is from system bus A or system bus B. When an invalidation address is provided to the invalidation queue FIFO and logic **34**, the queue control and bus spying logic **35** issues a Load-Queue signal via a line **39** as will be described in further detail with respect to FIG. **3**.

The cache memory system **21** further includes invalidation process control logic **40** to control the operations required in invalidating addresses in the cache module **30**. The invalidation queue FIFO and logic **34** issues an Invalidation Address Available signal to the invalidation process control logic **40** via a line **41** when the invalidation queue FIFO **34** is storing an invalidation address. The invalidation process control logic **40** issues an Unload-Queue signal to the invalidation queue FIFO and logic **34** via a line **42** to command the invalidation queue FIFO **34** to unload the invalidation address to the tag RAMs **32** via a bus **43**. The invalidation process control logic **40** communicates with the cache status indicators **33** via a bus **44** to determine if the tag RAMs **32** are busy with activity from the processor **20** or are available to perform invalidations and controls invalidation cycles through the bus **44**.

The invalidation queue FIFO and logic **34** issues a QFULL signal via a line **45** to the invalidation process control logic **40** when the invalidation queue FIFO **34** is logically full. The QFULL signal is also applied to the RETRY logic **25** via the bus **24**. The invalidation queue FIFO and logic **34** also receives adjustable upper-limit and lower-limit signals from the maintenance system **17** via the bus **26** to define when the invalidation queue FIFO is logically full and logically flushed, respectively, in accordance with the present invention, in a manner to be described.

Referring to FIG. **3**, in which like reference numerals indicate like components with respect to FIGS. **1** and **2**, details of the invalidation queue FIFO and logic **34** are illustrated. The invalidation queue is implemented by a FIFO **50** which functions as a buffer between the system buses **14** and the cache memory module tag RAM resources **32**. The invalidation queue mechanism is utilized since WRITE operations can occur on the system buses **14** more rapidly than the cache invalidation process can be executed. The queue FIFO **50** should be sufficiently wide to hold all of the pertinent information for each WRITE operation on the bus and adequately deep to hold sufficient WRITES to operate in accordance with the present invention, in a manner to be described. The depth of the FIFO **50** should be adequate to minimize the effect of invalidations on performance of the processor **20** and the computer system of FIG. **1** while assuring that no WRITES are lost. In the preferred embodiment of the invention, the FIFO **50** is 16 words deep to hold 16 invalidation addresses and 27 bits wide to hold a 26 bit address and 1 bit of block information.

An invalidation address is inserted into the FIFO **50** at word position **15** and an invalidation address is withdrawn from the FIFO at word position **0**. Preferably, the FIFO **50** is implemented as a very fast unclocked, ripple structure with incoming data rippling the entire depth of the FIFO **50** in approximately 5 nanoseconds. This speed is realized utilizing current MOS technology. Each invalidation address is inserted into the FIFO **50** at word position **15** and then ripples up the FIFO **50** from position **15** to the highest available empty position. If the FIFO **50** is empty, an

incoming invalidation address ripples up to word position 0 and is ready to be withdrawn from the queue.

A word is inserted into the FIFO 50 from a 27 bit bus 51 in response to the Load-Queue signal on the line 39 from the queue control and bus spying logic 35. A word is withdrawn from the FIFO 50 and applied to the 27 bit bus 43 in response to the Unload-Queue signal on the line 42 from the invalidation process control logic 40. Thus it is appreciated that the first word to enter the queue 50 is the first word withdrawn.

The invalidation queue FIFO and logic 34 further includes a Q-Counter 52 to track the number of invalidation addresses held in the FIFO 50 and to provide this number on counter output 53. The Q-Counter 52 receives a counter increment input from the Load-Queue signal on the line 39 and a counter decrement signal from the Unload-Queue signal on the line 42. Thus, each time address information is inserted into the queue FIFO 50, the Q-Counter is incremented by the Load-Queue signal and the Q-Counter 52 continues to indicate the current number of addresses held within the FIFO 50. Similarly, each time an invalidation address is withdrawn from the FIFO 50, the Unload-Queue signal decrements the Q-Counter 52 causing the counter to continue to hold the current number of addresses held in the FIFO 50.

In accordance with the invention, an upper-limit register 54 contains a QTRESH upper-limit threshold denoting when the FIFO 50 is logically full and a lower-limit register 55 contains a FLUSHLIM lower-limit threshold denoting when the FIFO 50 is logically flushed. In accordance with the invention, QTRESH and FLUSHLIM are adjustable and set from the maintenance system 17 via the bus 26 to optimize computer system performance.

The QTRESH and FLUSHLIM signals from the registers 54 and 55 are applied through comparators 56 and 57 to the set and reset inputs of a QFULL flip-flop 58, respectively. The comparators 56 and 57 also receive an input from the output 53 of the Q-Counter 52. The comparators 56 and 57 are configured such that the QFULL flip-flop 58 is set when the value in the Q-Counter 52 is greater than, or equal to the value in the upper-limit register 54 and is reset when the value in the Q-Counter 52 is less than, or equal to the value in the lower-limit register 55. The Q output of the flip-flop 58 provides the QFULL signal on the line 45 described above with respect to FIG. 2. The Q-NOT output of the flip-flop 58 provides a logical signal denoted as QNOTFULL on a line 59. In a manner to be described in greater detail below, when QFULL goes high, the FIFO 50 is flushed of invalidation addresses until QFULL goes low.

The invalidation queue FIFO and logic 34 further includes invalidation address available logic 60 that provides the Invalidation Address Available signal on the line 41 whenever an invalidation address resides in the FIFO 50 as discussed above with respect to FIG. 2. The logic 60 is responsive to the Load-Queue signal on the line 39, the Unload-Queue signal on the line 42 and the output 53 of the Q-Counter 52 for performing this function. Specifically, the Load-Queue signal on the line 39 sets a Queue Register Occupied Flip-flop (QROF) that generates the signal on the line 41. The flip-flop QROF is not shown. A combination of the Unload-Queue signal on the line 42 and the Q-Counter 52 going to zero resets QROF.

The invalidation queue FIFO and logic 34 includes internal spy logic 61 that provides interfaces to the system buses 14 through the external queue control and bus spying logic 35 (FIG. 2). The internal logic 61 includes an A-bus spy register 62, a B-bus spy register 63 and a multiplexer 64. The multiplexer 64 provides the invalidation address information to the FIFO 50 via the bus 51. The selection control input to the multiplexer 64 is provided by the bus-select signal on the line 38 from the queue control and bus spying logic 35. The inputs to the register 62 are provided by the system bus A signals on the path 36 and the inputs to the register 63 are provided by the system bus B signals on the path 37. The signals on the paths 36 and 37 are provided from the queue control and bus spying logic 35 as described above with respect to FIG. 2. The registers 62 and 63 function as system bus interfaces between system bus A and system bus B, respectively, and the FIFO 50 for the incoming system bus WRITE operation information.

The system bus A signals on the path 36 include an A-Load-Spy signal on a line 70, a 26 bit address field from the A-bus on a path 71 and a block bit on a line 72 indicating if the address field on the path 71 is for a single word WRITE or if it represents a multiple word block address. The A-Load-Spy signal on the line 70 loads the address and block information on the path 71 and line 72 into the register 62. In a similar manner, the system bus B signals on the path 37 are applied to the register 63 with respect to the system bus B WRITE operations. The path 37 is comprised of a line 73 for loading the register 63 in response to a B-Load-Spy signal, a 26 bit address path 74 and a line 75 for the block bit information.

Thus, the system address information is first held in the registers 62 and 63 and then multiplexed into the queue FIFO 50 via the multiplexer 64. The multiplexer select bit on the line 38 indicates whether the A or B bus is sourcing the address. The queue control and bus spying logic 35 (FIG. 2) selects which bus information is loaded into the FIFO 50 by controlling the multiplexer 64 via the bus-select signal on the line 38. If information from both system buses A and B arrive simultaneously, the A-Load-Spy signal on the line 70 and the B-Load-Spy signal on the line 73 concurrently load the bus information into the respective registers 62 and 63. Arbitration logic (not shown) in the queue control and bus spying logic 35 provides highest priority to the A-bus and services the B-bus on the next system clock period by appropriately controlling the bus-select signal on the line 38.

The enable inputs of the spy registers 62 and 63 receive the QNOTFULL signal from the flip-flop 58 on the line 59. In this manner the registers 62 and 63 are disabled during the flushing operation of the queue FIFO 50 when the QFULL signal on the line 45 is active. Thus, the spy registers 62 and 63 do not accept further invalidation addresses when QFULL is in effect.

Referring to FIG. 4, the queue level parameters that are adjustable in accordance with the present invention are graphically depicted. The "Upper-Limit Position" (b) is the level of the FIFO 50 where the QFULL condition occurs. The "Lower-Limit Position" (c) is the level of the FIFO 50 where the QNOTFULL condition occurs. The (b) level is denoted as QTRESH and the (c) level is denoted as FLUSHLIM. When the Upper-Limit Position (b) is reached, the FIFO 50 is considered logically full and a mandatory flush operation is initiated to flush the queue. The FIFO 50 is flushed until the Lower-Limit Position (c) is attained. This period (b–c) is denoted as the "Mandatory-Busy Period" during which the cache must execute mandatory invalidation processes. During the Mandatory-Busy Period, the cache module 30 is "busy" to the processor 20, as denoted by the cache status indicators 33 (FIG. 2).

The "Normal-Fill Region" (d–b) is the region of the FIFO 50 where the queue can fill and unfill under "normal"

conditions. The queue is not considered full in this region. Thus, in the Normal-Fill Region, the processor **20** and the invalidation process can alternate in accessing the cache tag RAM resources **32**.

A "Full-Fill Margin" (a–b) provides a significant safety margin region and encompasses the total number of additional incoming WRITE operations that can occur on the system buses without the loss of data coherency. The Full-Fill Margin is the space in the FIFO **50** above the position where QFULL occurs. The Full-Fill Margin is defined in accordance with the operational conditions of the computer system in which the invalidation queue mechanism of the present invention is utilized in a manner to be further explained.

In operation of the computer system of FIG. **1**, with continued reference to FIGS. **1–4**, when the central processor unit **10** operates in the Normal-Fill Region, the processor **20** and the invalidation process control logic **40** alternate in accessing the cache module **30**. The processor **20** utilizes the cache module **30** to perform data processing cycles and the invalidation process control logic **40** utilizes the cache module **30** to execute invalidation cycles. The processor **20** consults the cache status indicators **33** to determine if the cache module **30** is busy with the invalidation progress control logic **40** when the processor **20** desires to perform a data cycle. Conversely, the invalidation process control logic **40** consults the cache status indicators **33** to determine if the cache module **30** is busy with the processor **20** when the invalidation process control logic **40** has an invalidation cycle to perform.

In response to the Invalidation Address Available signal on the line **41**, the invalidation process control logic **40** performs an invalidation cycle by issuing the Unload-Queue signal on line **42** to withdraw the invalidation address from position **0** of the FIFO **50**. The invalidation address is transmitted to the tag RAMs **32** on the bus **43** to determine if the address is resident in the cache memory **31**. If so, the invalidation process control logic **40** marks the address invalid in the tag RAMs **32**.

Thus, the invalidation process control logic **40** will withdraw an invalidation address from the queue whenever one is in position **0** of the FIFO **50** and the cache tag RAM resources **32** are available. As discussed above, when an invalidation address is withdrawn from the FIFO **50**, the Q-Counter **52** is decremented. The described operations occur with respect to the Normal-Fill Region of FIG. **4** with the QNOTFULL signal on the line **59** from the QFULL flip-flop **58** high. With QNOTFULL high, the spy registers **62** and **63** are enabled.

The queue control and bus spying logic **35** monitors the system A and B buses **14** for WRITE operations and transmits the associated potential invalidation addresses along path **36** or **37** to the spy registers **62** or **63**, respectively, in accordance with which bus sourced the data. The queue control and bus spying logic **35** issues the appropriate A-Load-Spy or B-Load-Spy signal on the line **70** or **73** to load the spy register **62** or **63**, respectively. The multiplexer **64** is controlled by the bus-select signal on the line **38** to transmit the potential invalidation addresses to the FIFO **50** on the bus **51**. The potential invalidation addresses are loaded into position **15** of the FIFO **50** by the Load-Queue signal on the line **39** from the queue control and bus spying logic **35**.

As previously described, the addresses ripple through the FIFO **50** to the highest available position for ultimate withdrawal from position **0**. The invalidation queue is implemented utilizing the FIFO **50** so that from the addresses stored in the queue, the first address received will be the first address withdrawn for an invalidation cycle. The FIFO queue level indicator QTRESH marks how high the queue will be allowed to be filled before the QFULL condition occurs. The FLUSHLIM queue level indicator marks how low the FIFO queue will be shifted out before new invalidations are accepted by the spy registers **62** and **63**. Both limit registers **54** and **55** holding QTRESH and FLUSHLIM, respectively, are set during maintenance initialization.

When the queue FIFO **50** fills to position (b) of FIG. **4**, the invalidation process control logic **40** performs a mandatory invalidation process denoted as the Mandatory-Busy Period of FIG. **4**. The mandatory invalidation process is performed when the Q-Counter **52** attains the QTRESH value held in the upper-limit register **54**. When this occurs, the QFULL flip-flop **58** is set and the invalidation queue **50** is logically "FULL" with unloading of the addresses and the invalidation processes being mandatory.

When the invalidation queue **50** is full, the cache module **30** will go "busy" to the processor **20** as reflected by the cache status indicators **33**, thus rendering the cache tag RAM resources **32** available to the invalidation process on a full time basis. Accordingly, in response to QFULL, the invalidation process control logic **40** controls the mandatory invalidation process via the Unload-Queue signal on the line **42** and cache control signals on the bus **44**. The invalidation addresses are withdrawn from the queue FIFO **50** to the tag RAMs **32** and the invalidation processes occur at maximum speed. With each address withdrawal, the Q-Counter **52** is decremented. In the mandatory invalidation process, QNOTFULL is low thereby disabling the spy registers **62** and **63** so that during mandatory invalidation no new invalidation addresses are accepted. Additionally, QFULL is applied to the RETRY logic **25** (FIG. **1**) to retry WRITE requests on the system buses. The mandatory invalidation process is denoted as queue flushing.

The queue flushing operation continues until the Q-Counter **52** attains the FLUSHLIM value stored in the lower-limit register **55**. When this occurs, the queue FIFO **50** is logically NOT-FULL (logically empty) and the invalidation process control logic **40** controls the system to revert to the Normal-Fill Region operation described above with respect to FIG. **4**. When this logical NOT-FULL condition is attained, the mandatory invalidation process is no longer in effect, the cache module **30** is no longer busy to the processor **20**, and normal invalidations recommence as described above. When the Q-Counter **52** equals FLUSHLIM, the QFULL flip-flop **58** is reset enabling the spy registers **62** and **63** to once again accept invalidation addresses. Additionally, RETRY is no longer in effect.

In accordance with the invention, the queue structure **34** utilizes variable limits QTRESH and FLUSHLIM for the full and non-full indicators, respectively, of the FIFO **50**. By this mechanism, the system can be fine-tuned at initialization and debug time as to the queue level at which the mandatory-busy, queue flushing period will begin and also the queue level at which the mandatory invalidation condition will revert to the normal-fill, normal invalidation region. The fine tuning is achieved by empirically determining optimum values for QTRESH and FLUSHLIM and storing these values in the registers **54** and **55**, respectively. The values are determined by operating under maximum system conditions and evaluating system performance with various limit values. Preferably, the optimum values are loaded into the limit registers **54** and **55** at system initialization time.

The values QTRESH and FLUSHLIM are set to optimize system performance. Specifically, the limits should be cho-

sen to maximize the Normal-Fill Region while maintaining a sufficient Full-Fill Margin so as to prevent losing invalidation WRITES thereby maintaining data coherency. The limit values are also chosen to minimize RETRYing WRITE requests on the system buses. Operation in the Mandatory-Busy Period should be minimized to permit the processor **20** maximum access to the cache module **30** so as to maximize processor performance at the same time minimizing RETRYs which cause undesirable system bus traffic. Excessive RETRYing drastically slows down the operation of the system buses.

Although not discussed above nor shown in the Figures, it is appreciated that the bus interface of the I/O system **13** (FIG. 1) includes a RETRY counter. The RETRY counter times the RETRY wait interval for bus WRITE requests when activated by a RETRY signal on the system buses issued, for example, from the RETRY logic **25**. The values of QTRESH and FLUSHLIM should also be set so that the duration of the queue flushing operation coincides with a small number of RETRY wait intervals. The values of QTRESH and FLUSHLIM should be coordinated with the RETRY counter in the I/O system **13** which permits the I/O module to access the system buses after it has been RETRYed. If the gap between QTRESH and FLUSHLIM is large, but the RETRY counter value in the I/O system is low, the I/O system will be RETRYed frequently when the FIFO queue **50** is full. This results in unnecessary system bus traffic. Preferably, QTRESH and FLUSHLIM should be set to permit enough time for the QFULL flip-flop **58** to reset before allowing the I/O system **13** to access the system buses **14** after RETRY.

It is appreciated that there are literally hundreds of combinations of values that would provide varying degrees of performance in accordance with the configuration of the system. For example, in the above-described system utilizing the two spy registers **62** and **63** with bus write RETRYs implemented while the QFULL flip-flop **58** is set, a Full-Fill Margin of two would be adequate to maintain data coherency providing system data integrity. Room is provided in the queue for invalidation addresses that may be in the spy registers **62** and **63** when QFULL goes on. This consideration also enhances system performance by providing empty spy registers for immediate receipt of potential invalidation addresses from the system buses when the QFULL flip-flop **58** is reset. Although it is desirable to provide a large Normal-Fill Region, the Full-Fill Margin should remain sufficient to empty the spy registers when QFULL is enabled.

As discussed above, the spy registers **62** and **63** are disabled for receipt of invalidation addresses during the QFULL period. In architectures where the spy registers are not disabled and the bus WRITE requests are not RETRYed, a larger Full-Fill Margin would be required with concomitant adjustments to QTRESH and FLUSHLIM. Appropriate limits would be dynamically established to maximize processor performance under maximum system dynamic conditions.

In the system described above, an upper-limit of 8 and a lower-limit of 4 was established. These empirically derived limits resulted in system performance whereby over 90% of the time the queue FIFO **50** had only one or no invalidations pending in the queue under worst case data traffic conditions. Additionally, the processor **20** could access the cache resources over 90% of the time without the cache module **30** being busy because of invalidation processing. Furthermore, incoming invalidation addresses from the system buses were loaded into the queue without forcing system bus RETRYs of system WRITES over 90% of the time. With these limits, invalidations were performed in time slots between processor-to-cache accesses for over 90% of the computer system operating time.

In the above-described embodiment utilizing the two spy registers **62** and **63** and a queue FIFO **50** with a depth of 16 addresses, a QTRESH value of 13 with an appropriate FLUSHLIM value would also provide good performance. FLUSHLIM may be set at 11 or 12 so as to as quickly as possible return to normal operation.

The Full-Fill Margin is the safety margin accommodating the number of additional incoming WRITE operations that can occur on the system buses without the loss of data coherency or drastically slowing down the system buses by excessive RETRYing. If, for example, the upper-limit were set to the maximum size of the FIFO **50**, then when the mandatory unloading condition occurs, the queue would have a Full-Fill Margin of zero. There would be no room in the queue to accept additional incoming bus WRITES. If a new bus WRITE occurs before the first address can be withdrawn from the queue, a data coherency problem can occur resulting in questionable system integrity. On the other hand, if the upper-limit value is set too low, thereby providing a large Full-Fill Margin, then the invalidation queue **34** would enter the QFULL, Mandatory-Busy Period too often. Also, if the lower-limit value is set too far below the upper-limit value, the Mandatory-Busy Period would be of too long a duration. Since in the Mandatory-Busy Period no processor work is performed and the bus WRITES are continually RETRYed, these conditions could have significant performance effects.

The objective of fine-tuning the upper and lower invalidation queue limits is to create an adequately large Normal-Fill Region while providing sufficient Full-Fill Margin for safety. Additionally, the limit values are chosen to provide an appropriate queue flushing interval. The fine-tuning objective permits achieving the tag RAM operations of the processor retrieving hit data from the cache while system invalidations are performed without degrading system through-put and data integrity.

Preferably, the invalidation queue of the present invention is implemented in ASIC VLSI gate array hardware resulting in a large and yet relatively inexpensive queue to be utilized within the cache system. Normally, this gate array technology does not permit system hardware to be altered after initial design without significant schedule delay. By use of the adjustable queue system, as described above, fine-tuning after hardware design can be achieved to provide an optimum combination of performance and data integrity. The present invention provides for fine-tuning of queue size and operating characteristics after the gate array has been built and installed within the system. The invalidation queue has "soft" characteristics permitting adjustment in the queue upper and lower limits which specify when the queue is considered full and flushed, respectively. Utilization of the present invention facilitates the efficient processing of invalidation information at the least cost in performance to the computer system. The invention permits maximizing system memory bandpass for high speed data Input/Output traffic. Although the I/O traffic causes invalidations, the processor is not impeded thereby. Both the processor and the I/O subsystem can perform more work.

The present invention is explained above in terms of a storage queue with adjustable level limits. It is appreciated that the Full-Fill Margin of the present invention would also be useful for the reasons given above even with a fixed upper-limit position.

Although the invention is advantageously utilized as the invalidation queue for a cache memory system, the invention may also be advantageously utilized in other applications. For example, in a communication environment where packets of data are temporarily stored before transmission to a potentially busy receiving device, the logical size of the temporary storage and the level of flushing the store before accepting more data can be adjusted in accordance with dynamically varying conditions. In addition, the Full-Fill Margin can be utilized so as not to lose data.

The operation of the storage queue of the present invention was explained in terms of flushing the queue when the contents thereof attained the Upper-Limit Position and discontinuing the flushing operation when the contents attained the Lower-Limit Position. The queue might also be utilized in a mode whereby the queue is rapidly filled rather than flushed as controlled by the Upper-Limit and Lower-Limit Positions. In this mode, normal operation would be defined as (c)–(a), as illustrated in FIG. 4. When the contents of the queue attain the Lower-Limit Position, the queue is rapidly filled until the contents attain the Upper-Limit Position. Thereafter, the system is operated in a "normal" data transport mode.

Referring to FIG. 5, a bi-directional data communication system utilizing the adjustable queue of the present invention is schematically illustrated. Data is transported in full duplex fashion along a data communication medium **80**. The data passes through data transport and queue control logic **81** wherefrom the data may be diverted through adjustable queue and support logic **82** through full duplex bus **83**. The adjustable queue and support logic **82** is configured in a manner similar to that described above with respect to FIGS. **3** and **4**. The Upper-Limit Position adjustment is provided from an input **84** and the Lower-Limit Position adjustment is provided from an input **85**. Data entering the data transport and queue control logic **81** from the medium **80** is buffered in the adjustable queue **82** before being returned to the medium **80** by the logic **81**.

The Upper and Lower Limit Positions of the queue **82** are adjusted in the manner described above in accordance with the data transportation statistics and dynamics of the illustrated system. The queue **82** may be operated in either a rapid flushing or rapid filling mode, as described above, in accordance with system requirements. The data transport and queue control logic **81** controls the adjustable queue **82** via a path **86** in a manner similar to the embodiment of the invention described with respect to FIGS. **3** and **4**.

While the invention has been described in its preferred embodiment, it is to be understood that the words which have been used are words of description rather than limitation and that changes may be made within the purview of the appended claims without departing from the true scope and spirit of the invention in its broader aspects.

We claim:

1. In a computer system having first and second memory systems, said second memory system being a cache memory system for storing data resident in said first memory system, addressable locations of said first memory system being overwritten in operation of said computer system thereby creating overwritten addressable locations, said cache memory system comprising

spy means for monitoring when addressable locations of said first memory system are overwritten and for providing address signals representative of said overwritten addressable locations,

queue means responsive to said address signals for storing said address signals, thereby providing queued address signals, said queue means holding a number of said queued address signals,

invalidation means for withdrawing queued address signals from said queue means and marking locations of said cache memory system invalid in accordance therewith, said invalidation means being controllably operative to perform a queue flushing operation by continuously withdrawing queued address signals from said queue means and marking locations of said cache memory system invalid in accordance therewith,

upper limit determining means responsive to said number of said queued address signals for providing an upper limit signal when said number of said queued address signals reaches an adjustable upper limit,

lower limit determining means responsive to said number of said queued address signals for providing a lower limit signal when said number of said queued address signals reaches an adjustable lower limit,

said invalidation means being responsive to said upper and lower limit signals and operative to perform said queue flushing operation in response to said upper limit signal and to discontinue said queue flushing operation in response to said lower limit signal, and

setting means for setting said adjustable upper and lower limits,

said queue means having a maximum capacity,

said setting means being operative to set said adjustable upper limit at less than said maximum capacity so as to create a full-fill margin between said adjustable upper limit and said maximum capacity for accepting and storing address signals from said spy means when said queue flushing operation is being performed.

2. The cache memory system of claim **1** wherein said queue means comprises a FIFO.

3. The cache memory system of claim **1** further including a counter for providing a count signal representative of said number of said queued address signals,

said counter being responsive to said spy means for incrementing said count signal when an address signal is entered into said queue means,

said counter being responsive to said invalidation means for decrementing said count signal when a queued address signal is withdrawn from said queue means.

4. The cache memory system of claim **3** wherein said upper limit determining means includes comparator means responsive to said adjustable upper limit and said count signal for providing said upper limit signal when said count signal reaches said adjustable upper limit.

5. The cache memory system of claim **3** wherein said lower limit determining means includes comparator means responsive to said adjustable lower limit and said count signal for providing said lower limit signal when said count signal reaches said adjustable lower limit.

6. The cache memory system of claim **1** wherein said first memory system comprises a main memory of said computer system, said computer system comprising

a processor, said cache memory system being included in said processor,

an I/O system, and

bus means intercoupling said processor, said main memory and said I/O system.

7. The cache memory system of claim **6** wherein WRITE operations to said main memory are effected by issuing WRITE requests on said bus means, said processor further including

RETRY means responsive to said upper and lower limit signals for issuing a RETRY signal to said bus means

to cause RETRYing of said WRITE requests while said queue flushing operation is being performed.

8. The cache memory system of claim 6 wherein said setting means is operative for setting said adjustable upper and lower limits so as to minimize said issuing said RETRY signal to said bus means to enhance performance of said computer system.

9. The cache memory system of claim 8 wherein said setting means is operative for setting said adjustable upper and lower limits to 8 queued address signals and 4 queued address signals, respectively.

10. The cache memory system of claim 8 wherein said spy means comprises means for monitoring said bus means to detect said address signals representative of said overwritten addressable locations and to accept said address signals for transmission to said queue means for storage therein,

said spy means being further operative in response to said upper and lower limit signals to disable acceptance of said address signals from said bus means while said queue flushing operation is being performed.

* * * * *

US006249830B1

(12) **United States Patent**
Mayer et al.

(10) **Patent No.: US 6,249,830 B1**
(45) **Date of Patent: Jun. 19, 2001**

(54) **METHOD AND APPARATUS FOR DISTRIBUTING INTERRUPTS IN A SCALABLE SYMMETRIC MULTIPROCESSOR SYSTEM WITHOUT CHANGING THE BUS WIDTH OR BUS PROTOCOL**

(75) Inventors: **Dale J. Mayer**, Houston; **Sompong Paul Olarig**; **William F. Whiteman**, both of Cypress; **David F. Heinrich**, Tomball, all of TX (US)

(73) Assignee: **Compaq Computer Corp.**, Houston, TX (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/532,109**

(22) Filed: **Mar. 21, 2000**

Related U.S. Application Data

(63) Continuation of application No. 09/243,235, filed on Feb. 2, 1999, now Pat. No. 6,041,377, which is a continuation of application No. 08/699,912, filed on Aug. 20, 1996, now abandoned.

(51) **Int. Cl.**[7] **G06F 13/00**

(52) **U.S. Cl.** **710/113**; 710/260

(58) **Field of Search** 710/113, 9, 260

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,125,093		6/1992	McFarland	395/725
5,283,904	*	2/1994	Carson et al.	710/266
5,379,434		1/1995	DiBrino	395/725
5,410,710		4/1995	Sarangdhar et al.	395/725
5,434,997		7/1995	Landry et al.	395/575
5,437,042		7/1995	Culley et al.	395/800
5,446,910		8/1995	Kennedy et al.	395/800
5,481,725		1/1996	Jayakumar et al.	395/725
5,530,891		6/1996	Gephardt	395/800
5,555,430		9/1996	Gephardt et al.	395/800
5,564,060		10/1996	Mahalingaiah et al.	395/871
5,568,649		10/1996	MacDonald et al.	395/868
5,613,126		3/1997	Schmidt	395/733
6,041,377	*	3/2000	Mayer et al.	710/113

FOREIGN PATENT DOCUMENTS

0 602 858 A1	6/1994	(EP)	13/26
0 685 798 A2	12/1995	(EP)	13/26

OTHER PUBLICATIONS

Hewlett Packard; "White Paper on Multiprocessing"; Jun. 1995; pp. 1–11.

Intel; "Pentium Processor Integrated APIC"; pp. 19–4 through 19–40; Apr. 1994.

Advanced Micro Devices and Cyrix Corporation; AMD web site; "The Open Programmable Interrupt controller (PIC) Register Interface Specification Revision 1.2"; Oct. 1995; pp. 1–24.

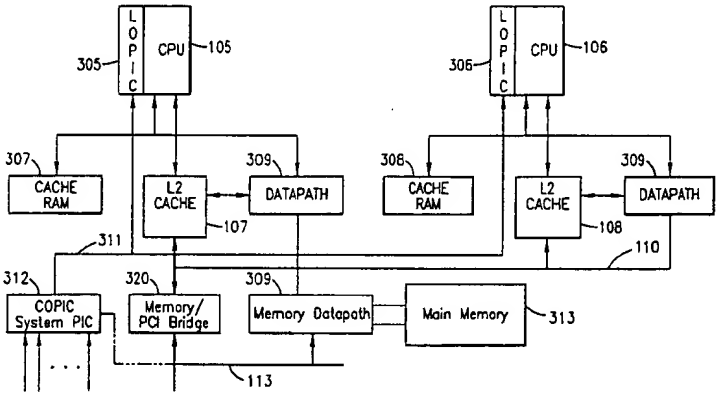
(List continued on next page.)

Primary Examiner—David A. Wiley
(74) *Attorney, Agent, or Firm*—Fletcher, Yoder & Van Someren

(57) **ABSTRACT**

A method for supporting multiple distributed interrupt controllers, designated as bus agents, in a symmetric multiprocessing system, which method includes the steps of assigning a unique identification number to each bus agent, receiving bus requests from the bus agents over four data lines in groups of four, and granting bus ownership to a selected one of the requesting bus agents. Similarly, a computer system that supports multiple distributed interrupt controllers, designated as bus agents, in a symmetric multiprocessing system, which computer system includes structure for assigning a unique identification number to each bus agent, four data lines for receiving bus requests from the bus agents in groups of four, and structure for granting bus ownership to a selected one of the requesting bus agents.

12 Claims, 9 Drawing Sheets



OTHER PUBLICATIONS

Cyrix Technical Brief; "OpenPIC™ Architecture"; pp. 1–2.
AMD News Release; "Advanced Micro Devices and Cyrix Jointly Develop OpenPIC™ Technology Standard"; Jun. 4, 1996; pp. 1–2.

Russo, A.P.: "The Alphaserver 2100 I/O Subsystem", Digital Technical Journal, vol. 6, No. 3, Jan. 1, 1994, pp. 20–28, XP000575573, p. 25, Right–Hand Column, Line 5–P. 26, Left–Hand Column, Line 46.

* cited by examiner

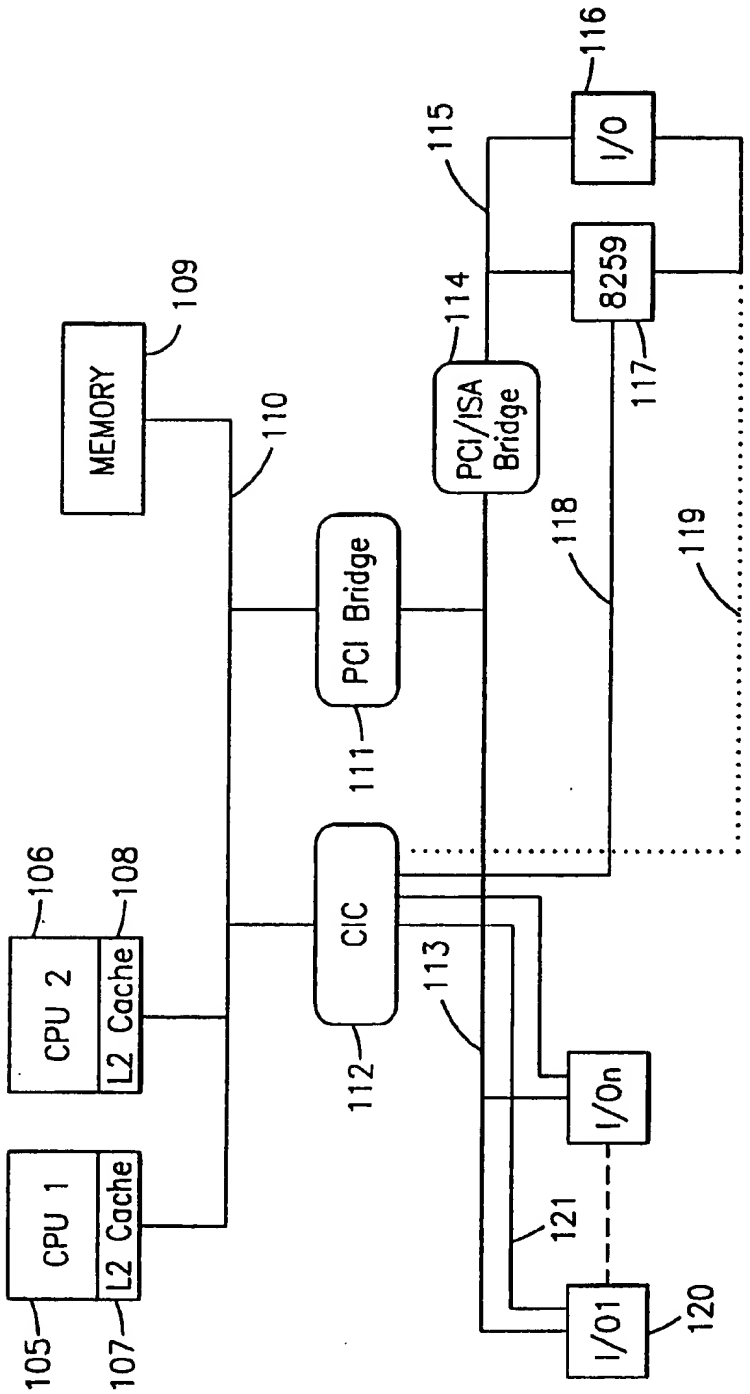


FIG. 1 (PRIOR ART)

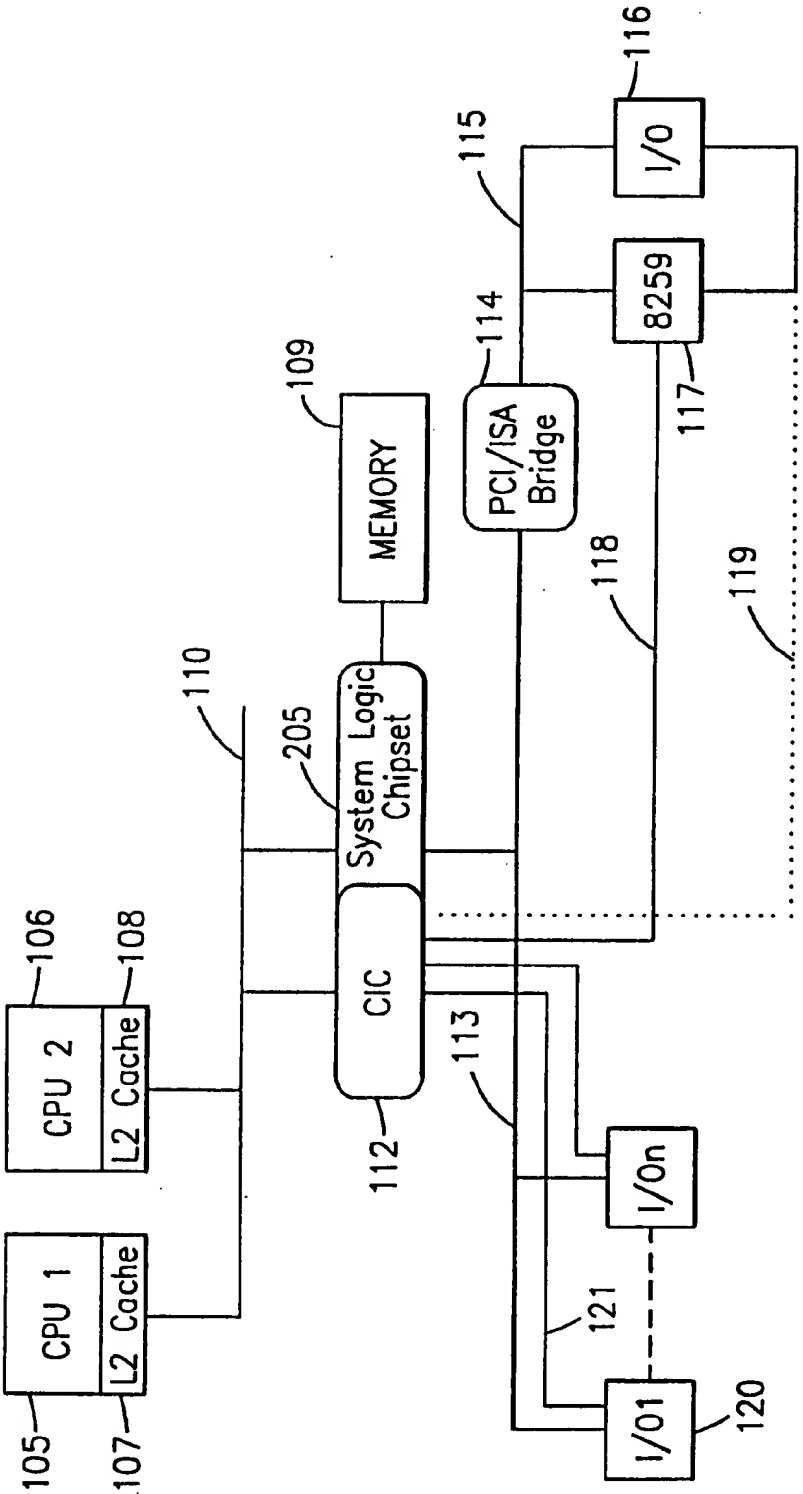


FIG. 2 (PRIOR ART)

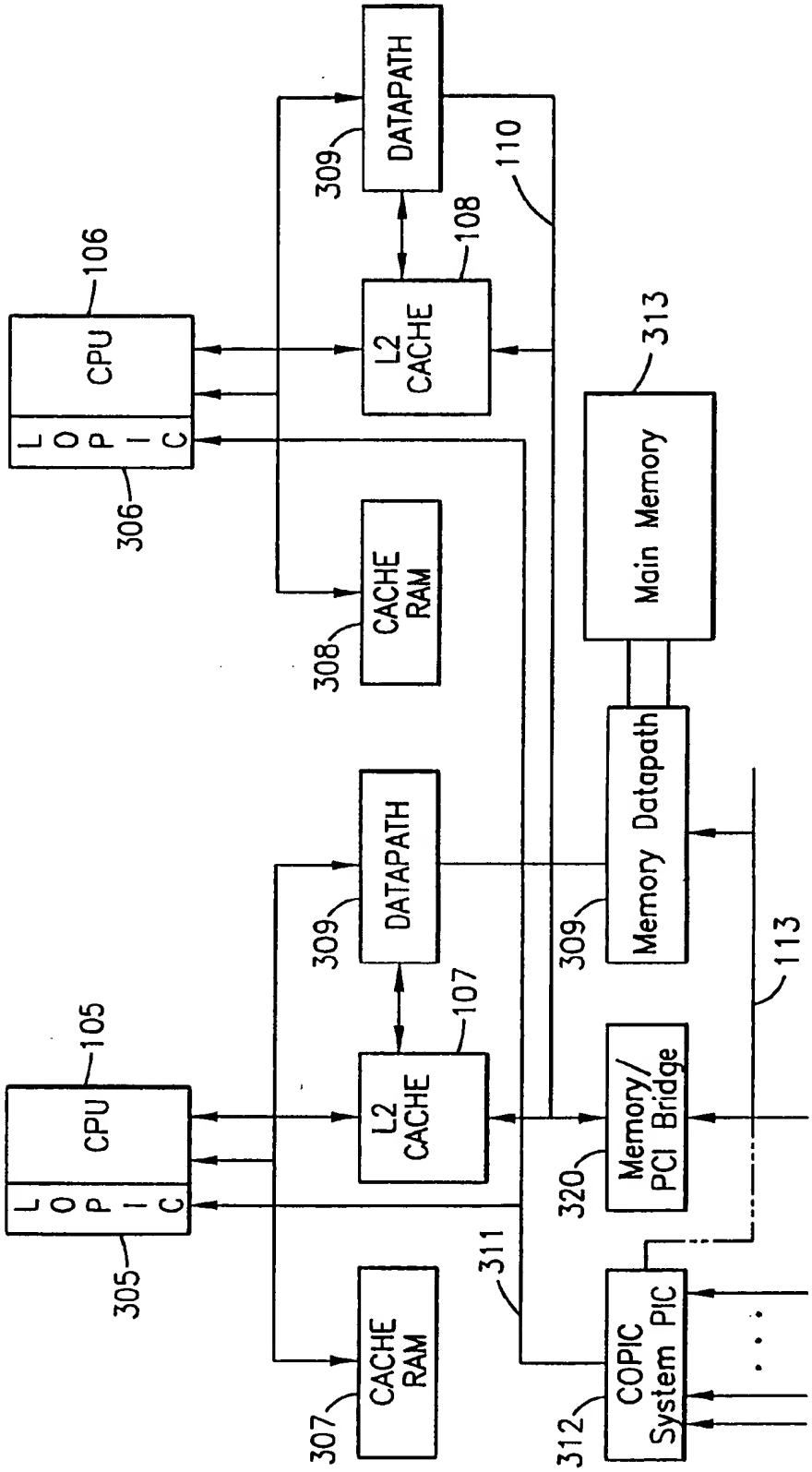

CPU
LOPIC
106
306
DATAPATH
309
110
108
L2 CACHE
CACHE RAM
308
313
Main Memory
Memory Datapath
309
DATAPATH
309
107
L2 CACHE
105
CPU
LOPIC
305
113
Memory/ PCI Bridge
320
311
CACHE RAM
307
COPIC System PIC
312

FIG. 3

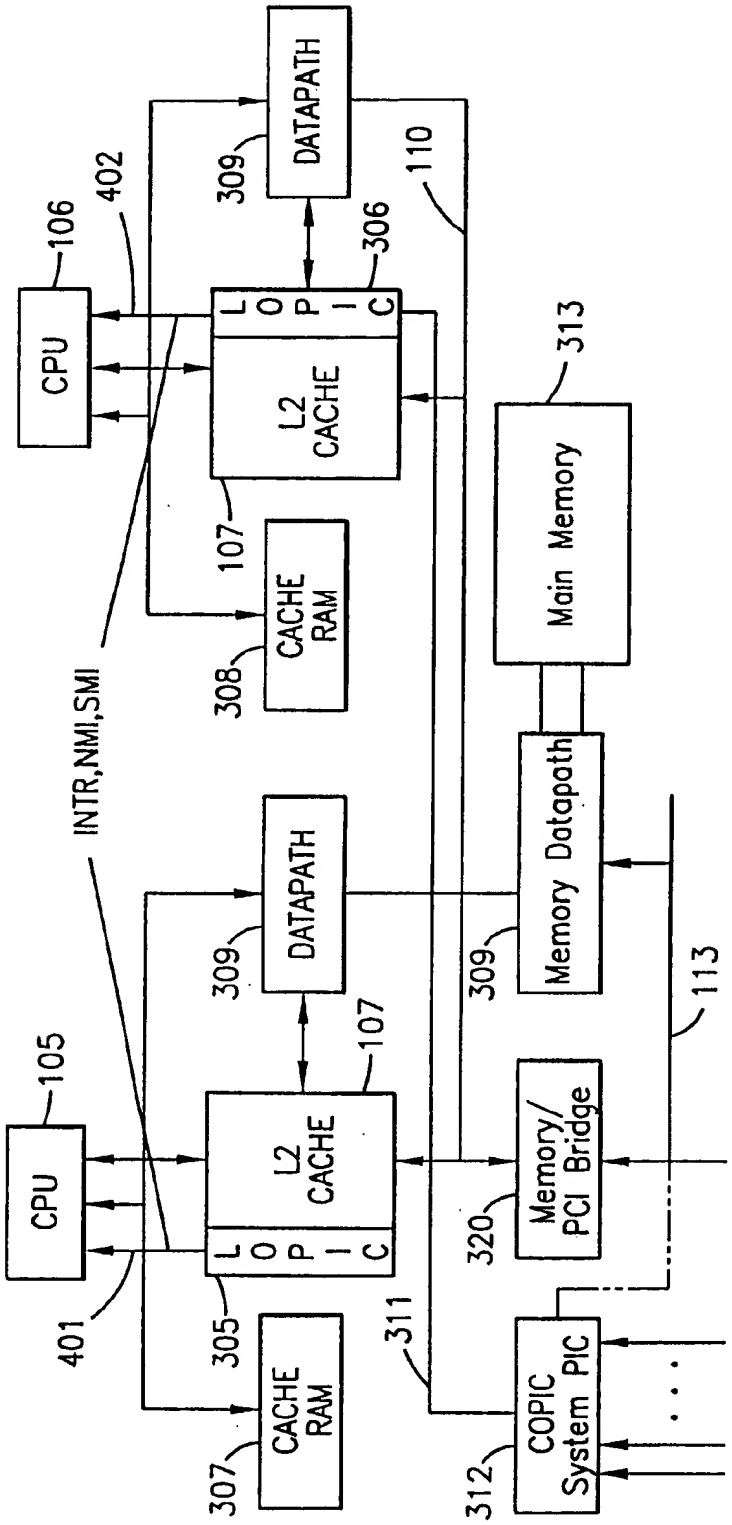

CPU
106
402
DATAPATH
309
110
306
L O P I C
L2 CACHE
107
CACHE RAM
308
INTR,NMI,SMI
313
Main Memory
Memory Datapath
309
113
DATAPATH
309
107
105
CPU
L2 CACHE
L O P I C
401
305
CACHE RAM
307
311
320
Memory/ PCI Bridge
COPIC System PIC
312

FIG. 4

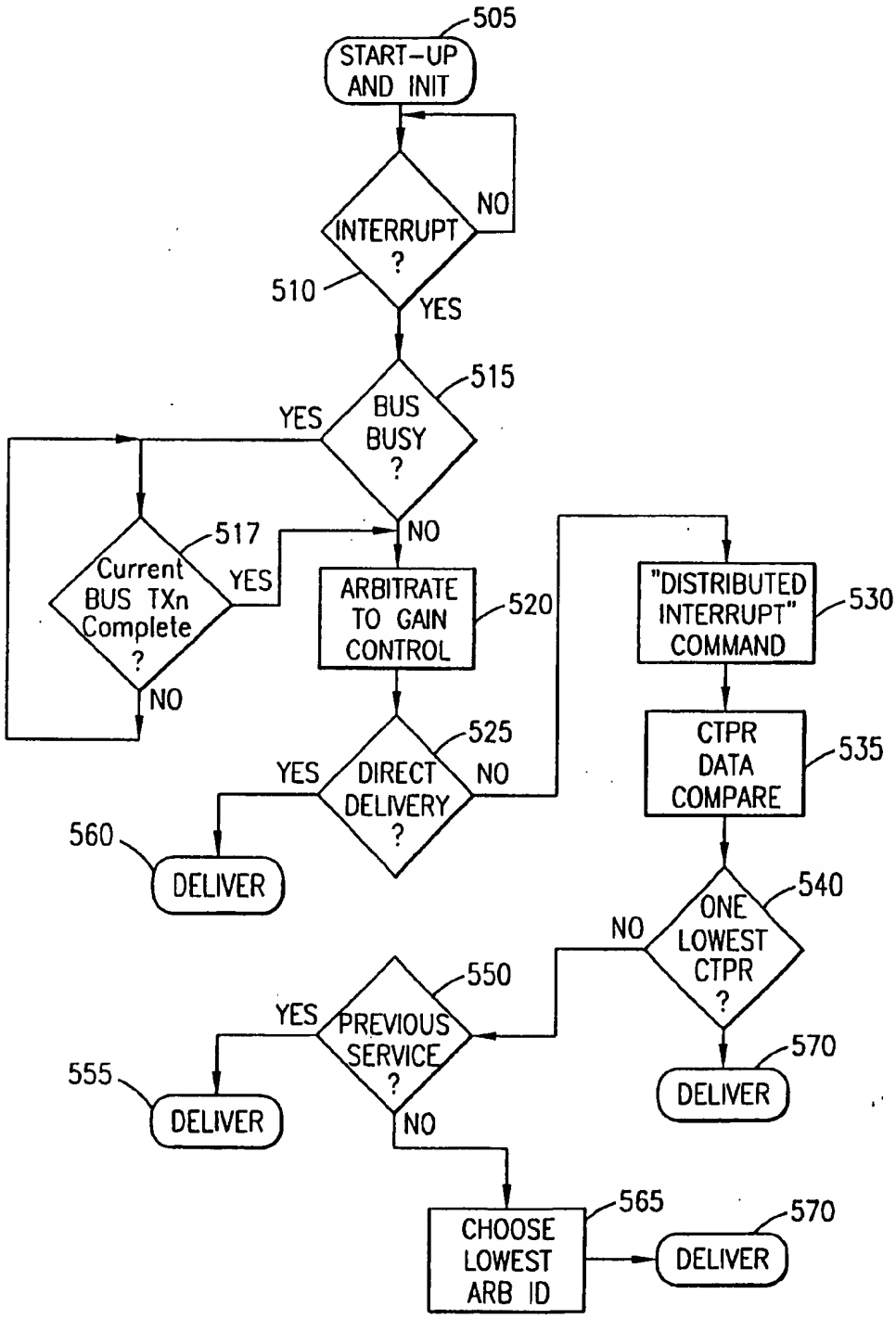

505
START-UP AND INIT
INTERRUPT ?
NO
510
YES
515
BUS BUSY ?
YES
NO
517
Current BUS TXn Complete ?
YES
NO
ARBITRATE TO GAIN CONTROL
520
525
DIRECT DELIVERY ?
YES
NO
560
DELIVER
"DISTRIBUTED INTERRUPT" COMMAND
530
CTPR DATA COMPARE
535
540
ONE LOWEST CTPR ?
NO
550
PREVIOUS SERVICE ?
YES
555
DELIVER
NO
570
DELIVER
565
CHOOSE LOWEST ARB ID
570
DELIVER

FIG. 5

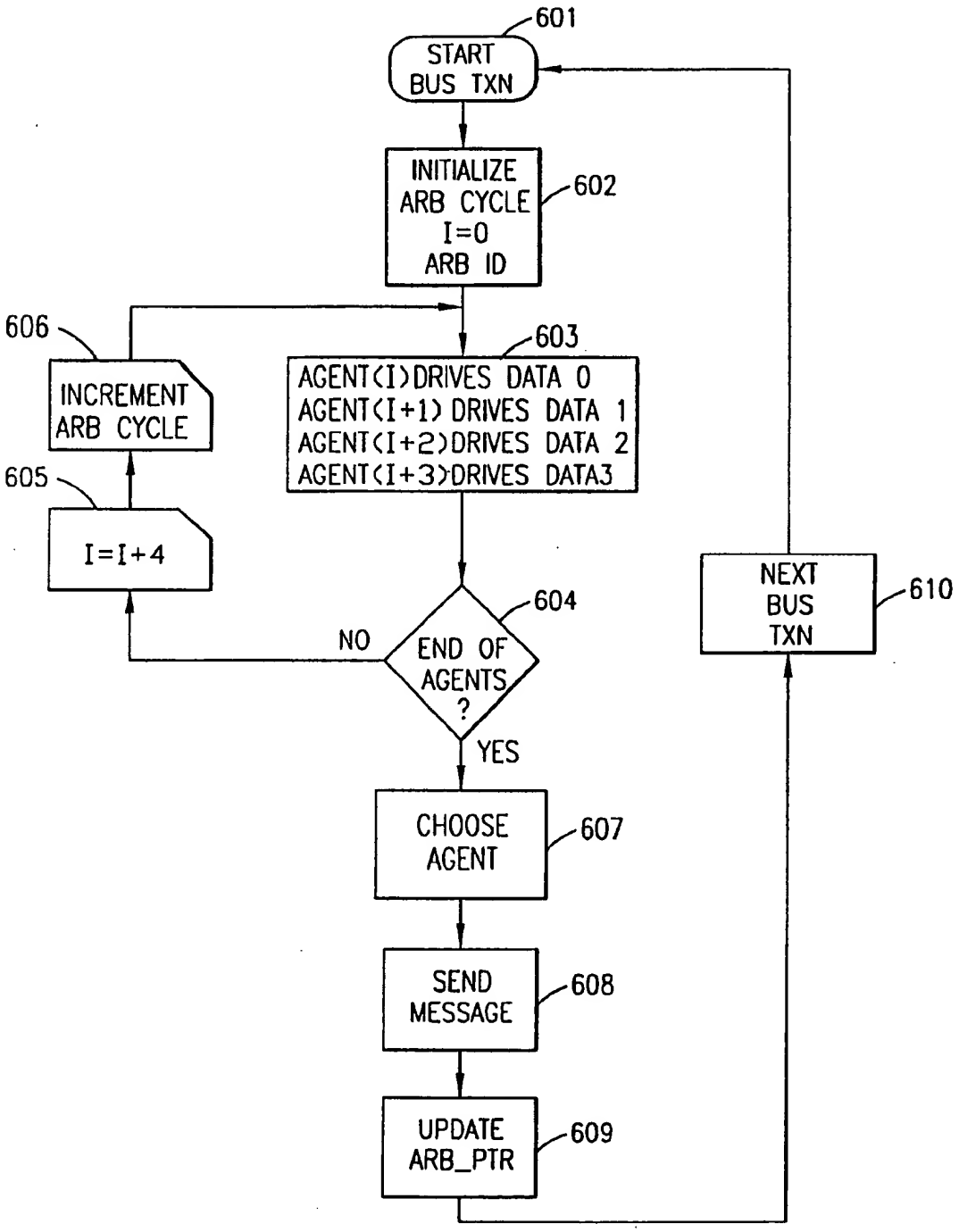

601
START BUS TXN
INITIALIZE ARB CYCLE I=0 ARB ID
602
606
INCREMENT ARB CYCLE
603
AGENT(I) DRIVES DATA 0
AGENT(I+1) DRIVES DATA 1
AGENT(I+2) DRIVES DATA 2
AGENT(I+3) DRIVES DATA3
605
I=I+4
604
NO
END OF AGENTS ?
YES
NEXT BUS TXN
610
CHOOSE AGENT
607
SEND MESSAGE
608
UPDATE ARB_PTR
609

FIG. 6

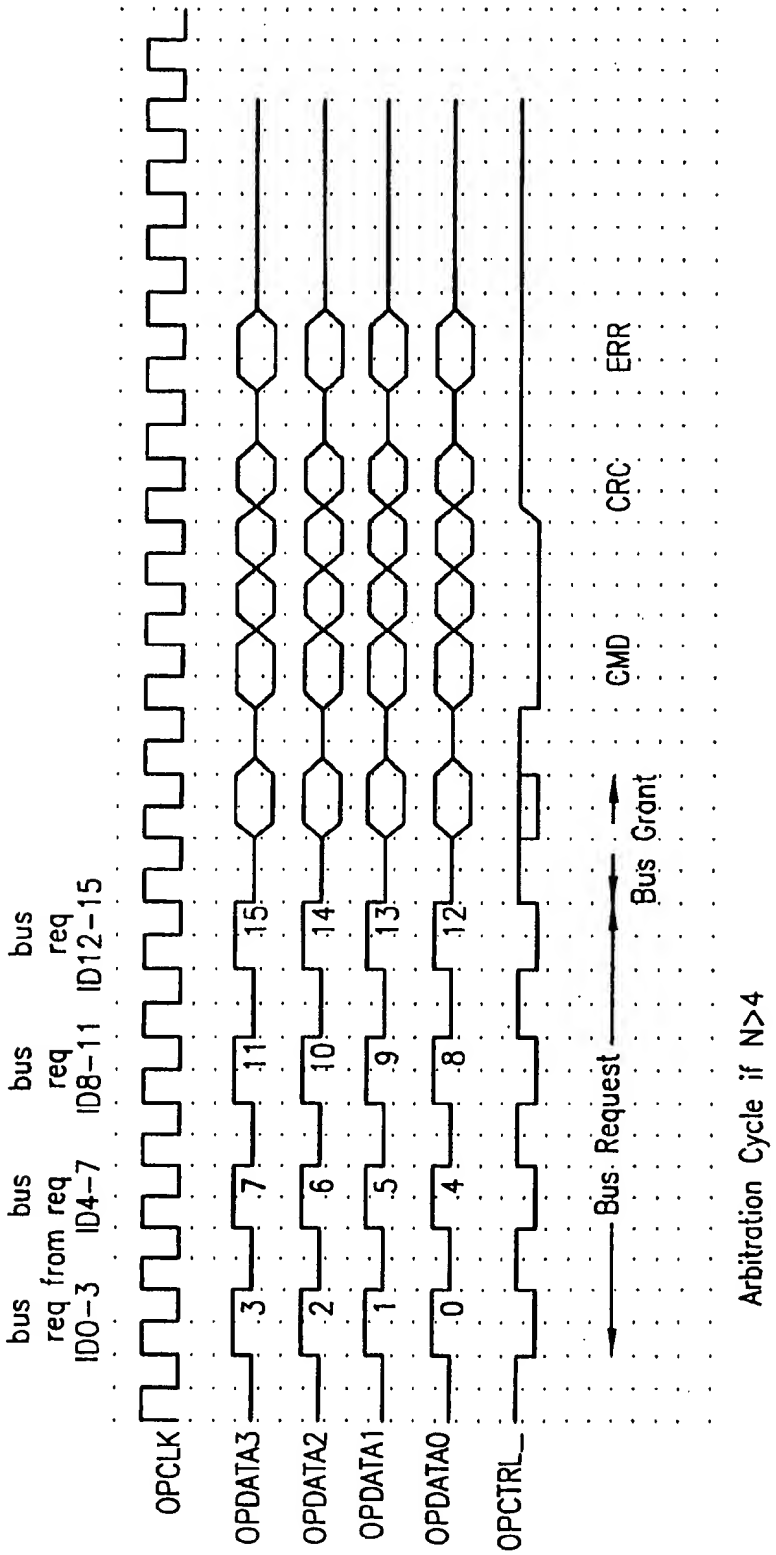

OPCLK
OPDATA3
OPDATA2
OPDATA1
OPDATA0
OPCTRL
bus req from req ID0–3
bus req ID4–7
bus req ID8–11
bus req ID12–15
3
7
11
15
2
6
10
14
1
5
9
13
0
4
8
12
Bus Request
Bus Grant
CMD
CRC
ERR
Arbitration Cycle if N>4

FIG. 7

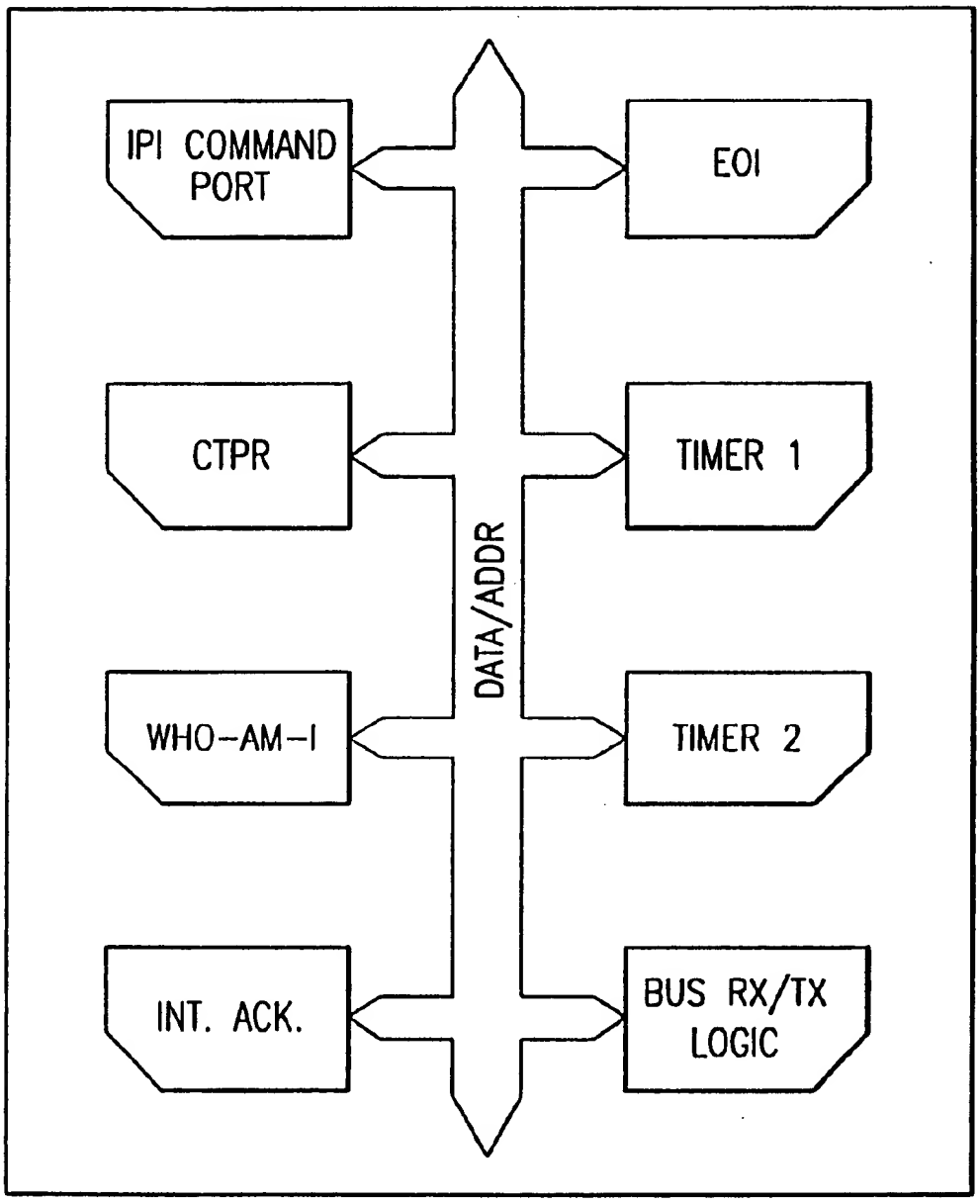

IPI COMMAND PORT
EOI
CTPR
TIMER 1
DATA/ADDR
WHO-AM-I
TIMER 2
INT. ACK.
BUS RX/TX LOGIC

FIG. 8

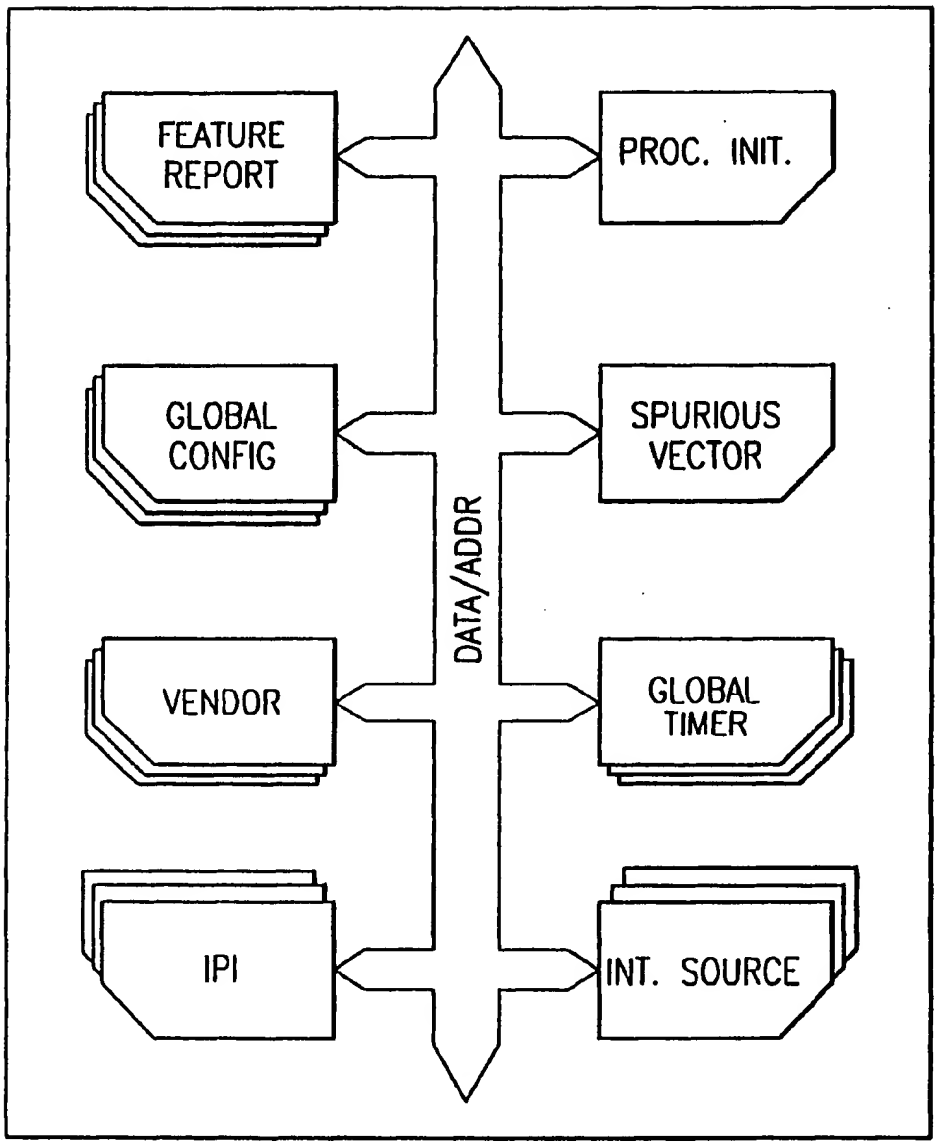

FEATURE REPORT
PROC. INIT.
GLOBAL CONFIG
SPURIOUS VECTOR
DATA/ADDR
VENDOR
GLOBAL TIMER
IPI
INT. SOURCE

FIG. 9

METHOD AND APPARATUS FOR DISTRIBUTING INTERRUPTS IN A SCALABLE SYMMETRIC MULTIPROCESSOR SYSTEM WITHOUT CHANGING THE BUS WIDTH OR BUS PROTOCOL

This application is a continuation of Ser. No. 09/243,235 filed on Feb. 2, 1999, now U.S. Pat. No. 6,041,377 which is a continuation of Ser. No. 08/699,912 filed on Aug. 20, 1996, now abandoned.

CROSS REFERENCE TO RELATED APPLICATION

This application hereby incorporates by reference the following co-assigned U.S. patent application, entitled "METHOD AND APPARATUS FOR DISTRIBUTING INTERRUPTS IN A SYMMETRIC MULTIPROCESSOR SYSTEM", concurrently filed herewith.

BACKGROUND OF THE INVENTION

1. Technical Field of the Invention

The present invention relates to computer systems and, in particular, to a method and apparatus for distributing interrupts in a scalable symmetric multiprocessor system.

2. Description of Related Art

The emergence of symmetric multiprocessing ("SMP") systems in today's high-end personal computer ("PC") and server markets has generated a need for new design approaches that achieve optimal performance within this expanded system structure. Some of the most significant challenges of multiprocessor system development include the design of a multiprocessor-capable bus ("MP bus") and the channeling and processing of interrupts through an SMP-aware interrupt controller. As is well understood in the art, the MP bus services multiple processing units, providing access to the main memory and other components of the system.

Conventionally, a multiprocessing system is a computer system that has more than one processor, and that is typically designed for high-end workstations or file server usage. Such a system may include a high-performance bus, huge quantities of error-correcting memory, redundant array of inexpensive disk ("RAID") drive systems, advanced system architectures that reduce bottlenecks, and redundant features such as multiple power supplies.

In the most general sense, multiprocessing is defined as the use of multiple processors to perform computing tasks. The term could apply to a set of networked computers in different locations, or to a single system containing several processors. As is well-known, however, the term is most often used to describe an architecture where two or more linked processors are contained in a single enclosure. Further, multiprocessing does not occur just because multiple processors are present. For example, having a stack of PCS in a rack is not multiprocessing. Similarly, a server with one or more "standby" processors is not multiprocessing, either. The term "multiprocessing", therefore, applies only when two or more processors are working in a cooperative fashion on a task or set of tasks.

There are many variations on the basic theme of multiprocessing. In general, the differences are related to how independently the various processors operate and how the workload among these processors is distributed. In loosely-coupled multiprocessing, the processors perform related

tasks, but, they do so as if they were standalone processors. Each processor may have its own memory and may even have its own mass storage. Further, each processor typically runs its own copy of an operating system, and communicates with the other processor or processors through a message-passing scheme, much like devices communicating over a local-area network. Loosely-coupled multiprocessing has been widely used in mainframes and minicomputers, but the software to do it is very closely tied to the hardware design. For this reason, it has not gained the support of software vendors, and is not widely used in PC servers.

In tightly-coupled multiprocessing, by contrast, the operations of the processors are more closely integrated. They typically share memory, and may even have a shared cache. The processors may not be identical to each other, and may or may not perform similar tasks. However, they typically share other system resources such as mass storage and input/output ("I/O"). Moreover, instead of a separate copy of the operating system for each processor, they typically run a single copy, with the operating system handling the coordination of tasks between the processors. The sharing of system resources makes tightly-coupled multiprocessing less expensive, and it is the dominant multiprocessor architecture in network servers.

Hardware architectures for tightly-coupled multiprocessing systems can be further divided into two broad categories. In symmetrical multiprocessor systems, system resources such as memory and disk input/output are shared by all the microprocessors in the system. The workload is distributed evenly to available processors so that one does not sit idle while another is loaded with a specific task. The performance of SMP systems increases, at least theoretically, for all tasks as more processor units are added. This highly sought-after design goal is called scalability.

In asymmetrical multiprocessor systems, tasks and system resources are managed by different processor units. For example, one processor unit may handle I/O and another may handle network operating system ("NOS") tasks. It can be readily seen that asymmetrical multiprocessor systems do not balance workloads. Thus, it is quite conceivable that a processor unit handling one task can be overworked while another unit sits idle.

It can further be noted that within the category of SMP systems are two subcategories, based on the way cache memory is implemented. The lower-performance subcategory includes "shared-cache" multiprocessing, and the higher-performance subcategory encompasses what is known as "dedicated-cache" multiprocessing. In dedicated-cache multiprocessing, every processor has, in addition to its "level 1" on-chip memory, a dedicated "level 2" off-chip memory cache (one per processor). These caches accelerate the processor-memory interaction in an MP environment. On the other hand, in shared-cache multiprocessing, the processors share a single "level 2" cache. Typically, shared-cache architecture offers less scalability than dedicated-cache architecture.

As briefly mentioned above, one of the most significant design challenges, in either broad category of multiprocessing, is the routing and processing of interrupts. Conventionally, an interrupt controller is responsible for delivering interrupts from interrupt sources to interrupt destinations in an MP system. An interrupt may be generalized as an event that indicates that a certain condition exists somewhere in the system that requires the attention of at least one processor. The action taken by a processor in response to an interrupt is commonly referred to as "servicing" or "handling" the interrupt.

In an SMP system, each interrupt has an identity that distinguishes it from the others. This identity is commonly referred to as the "vector" of the interrupt. The vector allows the servicing processor or processors to find the appropriate handler for the interrupt. When a processor accepts an interrupt, it uses the vector to locate the entry point of the handler in its interrupt table. In addition, each interrupt may have an interrupt priority that determines the timeliness with which the interrupt should be serviced relative to the other pending activities or tasks of the servicing processor.

There are, in general, two interrupt distribution modes available for an interrupt controller for delivering interrupts to their appropriate destinations in an MP system. In the directed delivery mode ("static" delivery), the interrupt is unconditionally delivered to a specific processor that matches the destination information supplied with the interrupt. Under the distributed delivery mode ("dynamic" delivery), interrupt events from a particular source will be distributed among a group of processors specified by the destination field value.

From the foregoing, it should be appreciated that balancing the interrupt loading among various processors in a scalable MP system is a very desirable goal. However, as is well understood in the art, it is a hard goal to accomplish from the system designer's perspective. Architecturally, two types of solutions exist for delivering interrupts to their destinations in an SMP environment. In one solution, for example, a centralized interrupt controller is disposed between a host bus and system bus such as the Peripheral Component Interconnect ("PCI") bus, for receiving interrupts from their sources and routing them to their destinations in either directed or distributed mode. Further, in this type of solution, a counter of certain bit-length is typically provided with each processing unit therein. The counter size is appended to a task priority register corresponding with a particular processing unit. The contents of the task priority register and the counter size appended thereto are used to determine the overall priority level for that processing unit. The counter associated with the processing unit is incremented, usually with a wraparound option, when an I/O interrupt is dispatched to that processing unit.

The second solution provides for a distributed interrupt control scheme wherein one interrupt controller assumes global, or system-level, functions such as, for example, I/O interrupt routing, while a plurality of local interrupt controllers, each of which is associated with a corresponding processing unit, control local functions such as, for example, interprocessor interrupts. Both classes of interrupt controllers may communicate through a separate bus, and are collectively responsible for delivering interrupts from interrupt sources to interrupt destinations throughout the system.

Both types of solutions described above are known to have several drawbacks. For example, in the centralized interrupt controller scheme of the foregoing, the width of the counter depends on the maximum number of processors allowed in the system, and because the width is appended to the task priority register of a processor, there is no guarantee that the selected processor for interrupt delivery in fact has the lowest priority, that is, it is the least busy unit, among the listed processors. In addition, since the scheme requires coupling of the centralized interrupt controller to the host bus and system bus, the interrupt messages will consume precious bandwidth on both buses, thereby negatively impacting the overall system performance. Further, the scalability of the centralized scheme will be degraded as more processors are added to the system.

On the other hand, although the distributed architecture has certain advantages, current distributed interrupt controller solutions also do not guarantee that the selected processor is indeed the one with the lowest priority as it is typically required that only those local interrupt controllers that have vacant interrupt slots be included in the lowest priority arbitration. Thus, it is possible that the selected processor might have the highest priority but is the only one that has at least one interrupt slot available.

A distributed interrupt controller solution that overcomes these and other disadvantages is described in greater detail in the co-assigned U.S. patent application, entitled "METHOD AND APPARATUS FOR DISTRIBUTING INTERRUPTS IN A SYMMETRIC MULTIPROCESSOR SYSTEM", cross-referenced hereinabove and concurrently being filed herewith. However, this cross-referenced patent application does not describe bus arbitration for a scalable scheme wherein multiple processors, typically in multiple sets of four processors—an architecture that is favored by digital design—are supported in a distributed interrupt controller system without changing the bus width or bus protocol.

Accordingly, it can be readily appreciated that there is a need for a cost-effective solution providing a scalable MP-compatible interrupt controller scheme that guarantees balanced delivery of interrupts to a processor that has the lowest current task priority among four or more processors without changing the bus width or bus protocol. Further, it would be advantageous to have such a scheme that is compatible with current industry architectures so as to maximize interoperability and interexchangeability. The present invention, described and claimed hereinbelow, provides a method and apparatus for accomplishing these and other objects.

SUMMARY OF THE INVENTION

A method for supporting multiple distributed interrupt controllers, designated as bus agents, in a symmetric multiprocessing system, which method includes the steps of assigning a unique identification number to each bus agent, receiving bus requests from the bus agents over four data lines in groups of four, and granting bus ownership to a selected one of the requesting bus agents.

A computer system that supports multiple distributed interrupt controllers, designated as bus agents, in a symmetric multiprocessing system, which computer system includes structure for assigning a unique identification number to each bus agent, four data lines for receiving bus requests from the bus agents in groups of four, and structure for granting bus ownership to a selected one of the requesting bus agents.

In one aspect of a presently preferred exemplary embodiment, the present invention provides a method of arbitrating bus control requests in a computer system of the type including one or more processing units, each of which is in communication with a corresponding bus agent, and a master arbiter with an internal arbitration pointer; the bus agent and the master arbiter being disposed on a programmable-interrupt-controller bus, the method comprising the steps of: initializing the contents of the internal arbitration pointer; asserting a bus control request signal by each bus agent, said asserting step being effectuated in groups of four if more than four bus agents are listed; and choosing a bus agent by the master arbiter based on an arbitration protocol, transmitting a bus message by the chosen bus agent, and updating the internal arbitration pointer to point to the chosen bus agent.

In another aspect, the present invention provides a computer system of the type including one or more processing

units, each of which is in communication with a cache memory unit, the computer system comprising one or more local programmable interrupt controllers, each of which local programmable interrupt controllers is disposed on a programmable-interrupt-controller bus, and at least one central programmable interrupt controller that is also disposed on the programmable-interrupt-controller bus, Each of the local programmable interrupt controllers comprises a current-task-priority register, an interprocessor-interrupt-command port, a who-am-i register, an interrupt-acknowledge register, an end-of-interrupt register, a first local timer, and a second local timer. Further, the central programmable interrupt controller comprises at least one feature-reporting register, at least one global-configuration register, at least one vendor-specific register, a vendor-identification register, a processor-initialization register, at least one interprocessor-interrupt-vector-priority register, a spurious-vector register, at least one global-timer register, and at least one interrupt source register. The exemplary computer system of the present invention also includes a host bus, the host bus being disposed among the cache memory units for providing a communication path therebetween; a first system bus, the first system bus being coupled to the host bus through a first bus-to-bus bridge; and a second system bus, the second system bus being coupled to the first system bus through a second bus-to-bus bridge. Also, a standard 8259A-compatible interrupt controller may be provided on the second system bus such that start-up power-on compatibility is included. Additionally, the programmable-interrupt-controller bus comprises six electrically conductive transmission lines, one clock, one control and four data lines. In a further embodiment, each of the local programmable interrupt controllers may be integrated with its corresponding processing unit.

In a yet another aspect, the present invention includes a method of delivering interrupts in a scalable multiprocessor computer system of the type including a master arbiter residing in a central interrupt controller and one or more bus agents, the central interrupt controller and the bus agents being disposed on a programmable-interrupt-controller bus, the method comprising the steps of: determining the presence of an interrupt; gaining control of the programmable-interrupt-controller bus; determining status as to whether the interrupt is a directed delivery interrupt; delivering the interrupt to a pre-specified processor if the interrupt is determined to be a directed delivery interrupt in response to the step of determining status; otherwise, selecting the unit having the lowest value in its current-task-priority register; and delivering the interrupt thereto in response to the selecting step.

In a further embodiment, the gaining control step further comprises the steps of: asserting a bus request signal by a bus agent; choosing the bus agent by the master arbiter wherein the bus agent is pointed to by an internal arbitration pointer; transmitting a bus message by the chosen bus agent; and incrementing the contents of the arbitration pointer with wraparound.

In a yet another embodiment, the selecting step further comprises the steps of: sending a distributed-interrupt command to the listed bus agents by the master arbiter; transmitting the current task priority level data serially on the programmable-interrupt-controller bus by the listed bus agents in response to the distributed-interrupt command; and determining which listed bus agent has the lowest current task priority level. In this embodiment of the invention, also: if only one bus agent has the lowest current task priority level, then choosing that bus agent for delivery of the interrupt thereto; otherwise, choosing the bus agent that has previously serviced the interrupt for interrupt delivery. Still further in this embodiment of the invention, if no agent has previously serviced the interrupt, then the one with lowest bus identification value is chosen for delivery.

BRIEF DESCRIPTION OF THE DRAWINGS

A more complete understanding of the method and apparatus of the present invention may be had by reference to the following Detailed Description when taken in conjunction with the accompanying Drawings wherein:

FIG. **1** is a block diagram of a conventional multiprocessor computer system having a centralized interrupt controller;

FIG. **2** depicts a block diagram of a second embodiment of the conventional multiprocessor computer system having a centralized interrupt controller;

FIG. **3** depicts a block diagram of a first embodiment of the present invention directed to a scalable multiprocessor computer system having a distributed interrupt control scheme wherein four or more processing units are disposed;

FIG. **4** depicts a block diagram of a second embodiment of the present invention;

FIG. **5** illustrates an exemplary flowchart for delivering interrupts in a multiprocessor computer system having a distributed interrupt control scheme in accordance with the teachings of the present invention;

FIG. **6** illustrates an exemplary flowchart for bus arbitration in a scalable multiprocessor computer system including four or more processing units;

FIG. **7** depicts an exemplary timing diagram for bus arbitration with multiple arbitration cycles in accordance with the teachings of the present invention;

FIG. **8** illustrates an exemplary embodiment of a local programmable interrupt controller in accordance with the teachings of the present invention; and

FIG. **9** illustrates an exemplary embodiment of a central programmable interrupt controller in accordance with the teachings of the present invention.

DETAILED DESCRIPTION OF DRAWINGS

Referring now to the drawings wherein like or similar elements are designated with identical reference numerals throughout the several views, and wherein the various elements depicted are not necessarily drawn to scale, and in particular to FIG. **1**, there is depicted a block diagram of a conventional multiprocessor computer system having a centralized interrupt controller **112**. The multiprocessor computer system comprises a plurality of processing units, of which two processors are shown, labeled CPU1 and CPU2 and designated with reference numerals **105** and **106**, respectively.

Each of the processors, **105** and **106**, is coupled to a corresponding cache memory unit, designated with reference numerals **107** and **108**, respectively. A system memory **109** is in communication with the processor/cache combination units via a host bus **110**. The centralized interrupt controller (labeled "CIC") **112** is connected to the host bus **110** and a first system bus **113**, such as a peripheral component interconnect bus. The first system bus **113** is bridged to the host bus **110** via a first bus-to-bus bridge, labeled PCI bridge and designated with reference numeral **111**.

Continuing to refer to FIG. **1**, the first system bus **113** is also bridged to a second system bus **115**, such as an industry

standard architecture ("ISA") bus, via a second bus-to-bus bridge, labeled PCI/ISA bridge and designated with reference numeral **114**. Disposed on the first system bus **113** are a first plurality of I/O devices, one of which devices is referenced with reference numeral **120**, that may function as I/O interrupt sources. As shown herein, each of these I/O devices is connected to the CIC **112** via its own IRQ path, such as, for example, IRQ line **121**.

Still continuing to refer to FIG. **1**, disposed on the second system bus **115** are a second I/O block **116** and a conventional 8259 interrupt controller **117**. The second I/O block **116** may also function as an interrupt source to the multiprocessor computer system. The output from the 8259 interrupt controller **117** is connected via an IRQ line **119** to the CIC **112** to enable the pass-through mode for start-up operations.

In general operation, after power-on reset, the conventional CIC **112** will default to the 8259 pass-through mode. In this mode, the 8259 interrupt request output will be passed directly through the CIC **112** to a single, pre-selected processor's interrupt request input line and the CIC **112** will be essentially disabled. During SMP operation, the 8259 pass-through mode will be disabled, and the CIC 112 will distribute all system interrupt events, as described immediately hereinbelow.

Each processor, for example CPU1 **105**, is provided in a 4-processor exemplary implementation with a two-bit counter (not shown) that is initialized to either 00, 01, 10, or 11. Each of these two-bit counters is appended to a four-bit task priority register (not shown) associated with a processor, for example CPU1 **105**. Therefore, each processor essentially has a six-bit internal priority level, and as can be readily seen, even if all four-bit task priority registers contain the same priority data, each of the four processors will have a different six-bit internal priority level. When an I/O interrupt is dispatched to any of the processors, each processor's two-bit counter is incremented by one (or wrapped around, if necessary). This causes the processor that had the lowest six-bit internal priority level before its counter is incremented to not be the lowest in priority after the increment operation.

As mentioned hereinabove, this implementation does not guarantee that the selected processor will have the lowest four-bit priority level. In addition, since this system requires that the CIC **112** be attached to both the host bus **110** and the first system bus **113**, the interrupt messages will congest and consume bus bandwidth on both buses. As can be readily appreciated, by increasing the number of processors in the system, the bus traffic for interrupt control will only exacerbate the situation, thereby compromising system scalability.

Referring now to FIG. **2**, therein is shown a block diagram of a second embodiment of the conventional multiprocessor computer system having a centralized interrupt controller, described immediately above in reference to FIG. **1**. Essentially, the embodiment in FIG. **2** can be seen to be very similar to the embodiment depicted in FIG. **1**. In the embodiment of FIG. **2**, however, the CIC **112** is integrated into a system logic chipset **205** such that it has internal connection capability to both the host bus **110** and the first system bus **113**. Although this embodiment may provide certain advantages over the embodiment of FIG. **1**, such as, for example, lower cost and pin count, it is still burdened with the disadvantages, such as, for example, consumption and congestion of the host/system buses; uncertainness in the priority of the selected processor; and less-than-optimal scalability, discussed hereinabove.

FIG. **3** depicts a block diagram of a first exemplary embodiment of the present invention directed to a scalable multiprocessor computer system having a distributed interrupt control scheme. The exemplary multiprocessor computer system may have one or more processors, two of which are labeled and designated as CPU **105** and CPU **106**. Each of the processors, for example, CPU **105** or CPU **106**, is coupled to a corresponding local programmable interrupt controller, for example, LOPIC **305** or LOPIC **306**, respectively. According to the teachings of the present invention, each LOPIC handles interrupt delivery protocol with its corresponding processor. The LOPIC also handles accesses to the processor's internal registers, inter-processor interrupts ("IPI") and remote accesses. In addition, each LOPIC can be disabled by hardware, or software. Each LOPIC may be used in conjunction with a standard 8259A-compatible interrupt controller (not shown in this FIG.) for start-up pass-through mode capability. In a presently preferred embodiment, each LOPIC contains two local timers (not shown) which may be used by the system software to monitor system performance and/or for diagnostic purposes. Further, each LOPIC contains a interprocessor-interrupt-command port (not shown), writing to which port will cause an IPI to be sent to one or more processors; and a current-task-priority register ("CTPR") which is used for setting the current task priority of each processor. The task priority indicates the relative importance of the currently executing task. In this exemplary embodiment, priority levels from 0 to 15 are implemented.

Continuing to refer to FIG. **3**, each LOPIC, for example LOPIC **305**, further contains a who-am-i register which provides a mechanism for each processor, for example, CPU **105**, to determine the ID value of that processor. This value may be used to determine the value of the destination masks used for delivering interrupts. The LOPIC **305** also contains an interrupt-acknowledge register; and an end-of-interrupt ("EOI") register, writing a zero to which register will signal the end of processing for the interrupt currently in progress for the associated processor.

FIG. **8** depicts an exemplary embodiment of LOPIC **305** and its contents, which contents may be implemented and interconnected conventionally to allow execution of LOPIC functions in embodiments of the present invention.

Referring again to FIG. **3**, each processor, for example, CPU **105**, is in data communication with an associated cache memory unit, for example, cache **107**. There may also be secondary cache; for example, cache RAM **307** and cache RAM **308**; that is connected via a data path **309** to a main memory **313**. Cache **107** and cache **108** are electrically connected to a host bus **110** which is bridged via a first bus-to-bus bridge, bridge **320**, to a first system bus **113**, such as a PCI bus. Although not shown in this FIG., it can be appreciated that a second system bus, for example, an EISA or ISA bus may be bridged to the bus **113** via a suitable bus-to-bus bridge.

Still continuing to refer to FIG. **3**, each LOPIC, for example LOPIC **305** or LOPIC **306**, is connected via a programmable-interrupt-controller bus **311** to a central programmable interrupt controller, COPIC **312**. Although only one COPIC **312** is shown in this embodiment, it can be readily understood upon reference hereto that a plurality of central programmable interrupt controllers may be used in a system constructed according to the teachings of the present invention. In an SMP environment, multiple LOPIC and COPIC units operate together and are collectively responsible for delivering interrupts from interrupt sources to interrupt destinations throughout the system. The COPIC

312 is connected to a plurality of I/O interrupt sources (not shown), and provides functions such as I/O interrupt routing, masking of interrupts and bus arbitration. The presently preferred embodiment of COPIC **312** contains global registers such as feature-reporting registers, global-configuration registers, vendor-specific registers, a vendor-identification register, a processor-initialization register, IPI-vector/priority registers, a spurious-vector register, global-timer registers, and interrupt-source registers.

FIG. **9** depicts an exemplary COPIC **312** and its contents, which contents may be implemented and interconnected conventionally to allow execution of COPIC functions in embodiments of the present invention.

Referring again to FIG. **3**, the programmable-interrupt-controller bus **311** is preferably a six-wire, multi-phase, bi-directional shared bus having four data lines, one control line, and a clock line. The bus **311** is preferably optimized at present for a four-processor environment. However, as will be seen hereinbelow in specific reference to FIGS. **5**, **6** and **7**, the bus arbitration protocol is amenable in accordance with the teachings of the present invention to supporting more than four processors without having to change the bus width, thereby maintaining scalability. In a system with multiple COPIC and LOPIC units, one COPIC will be a priori designated as a master/arbiter and the non-master COPICs and the LOPICs (which may be integrated with their respective processing units, as shown in this FIG.) will be treated collectively as bus agents. The general operation of bus arbitration and interrupt delivery will be discussed below in specific reference to FIGS. **5**, **6**, and **7**.

Referring now to FIG. **4**, therein is depicted a block diagram of a second embodiment of the present invention directed to a multiprocessor computer system having a distributed interrupt control scheme. This embodiment is similar to the one discussed in detail hereinabove in reference to FIG. **3**. However, the LOPIC units, **305** and **306**, are provided in this embodiment as external units that are in communication with their corresponding processors, CPU **105** and **106**, via external paths **401** and **402**, respectively, rather than as integrated units with processors having internal access thereto. It can be readily understood that the exemplary embodiment discussed in reference to FIG. **3** provides a better solution in terms of lower cost and lower pin count than the one described in this FIG., but this design is an alternative design that may be usefully implemented in some applications.

FIG. **5** is an exemplary flowchart for delivering interrupts in an SMP environment in accordance with the teachings of the present invention. During system power up and initialization step **505**, each bus agent (discussed in reference to FIG. **3**) is assigned a unique arbitration identification ("Arb ID") value. Because of the 8259A-compatibility, the power up reset will have been processed by a prespecified processor in the system. In addition, each bus agent will be initialized as to the total number of bus agents on the programmable-interrupt-controller bus **311** (shown in FIG. **3**). The preferred method is to allow the system software ("OS") to control and provide all the necessary information during the system initialization phase **505**. In order to maintain scalability, the preferred embodiment employs an encoded 5-bit register wherein a value of "00000" indicates the presence of one bus agent and one master COPIC; a value of "00001" indicates two bus agents and one master COPIC; and so on, up to a value of "11110" indicating 31 bus agents and one master COPIC. Additionally, a value of "11111" in the encoded 5-bit register indicates a RESET in the exemplary embodiment.

If the system detects the presence of an interrupt, as provided in the decision block **510**, then either a COPIC or a LOPIC must gain control of the programmable-interrupt-controller bus **311** before a bus message may be delivered therethrough. If the bus **311** is busy, as determined in the decision block **515**, then a bus agent must wait until the bus **311** is idle, as indicated in the decision block **517**. Then, once the current bus transaction is complete, the bus agent must arbitrate to gain a bus grant, since only the master/arbiter COPIC will have total control of its bus request line which is connected to the control line of the bus. The arbitration phase **520** will be discussed in greater detail hereinbelow in specific reference to FIGS. **6** and **7**, taken together. If the bus **311** is idle, an agent or the master may directly start an arbitration cycle to gain a bus grant.

Upon entering the decision block **525** after gaining control of the bus **311**, a determination is made if the interrupt is a directed delivery mode interrupt or a distributed delivery mode interrupt. If the determination is that the interrupt is a directed delivery mode interrupt, then by taking the YES path therefrom, data is delivered to the destination processor (or, bus agent/LOPIC, if the processor and its LOPIC are not integrated together) on a plurality of data lines, for example, four data lines, that comprise a portion of the bus **311**.

If the determination in step **525** is that the interrupt is a distributed delivery mode interrupt, then a COPIC will send a "distributed interrupt" command to all listed agents, as provided in step **530**. Each agent then provides its current task priority level (CTPR) by transmitting its four-bit priority level serially via the data line that is used to request the bus with. Because the number of bits in a CTPR level is four, a total of four cycles is required for each agent to provide its CTPR value to the COPIC.

After receiving the CTPR values, the COPIC compares and selects in step **535** the agent that has the is lowest CTPR value (0 is the lowest, meaning the least busy; 15 is the highest). If only one agent has the lowest CTPR value, then by taking the YES path from decision block **540**, the interrupt is delivered to that agent. Otherwise, a determination is made in decision block **550** to select the agent that previously serviced the same interrupt for delivery thereof as provided in step **555**. If, on the other hand, no agent previously serviced that interrupt, then a choice is made in step **565** to select the agent based on its unique Arb ID, for example, selecting the one with the lowest Arb ID for delivering the interrupt thereto.

Referring now to FIGS. **6** and **7** together, a scheme for bus arbitration in an exemplary scalable MP system having four or more processing units (each of which may be integrated with its respective LOPIC) is described in a flowchart in FIG. **6** and as a bus timing diagram in FIG. **7**. As can be readily appreciated, the same bus arbitration scheme may be used in an embodiment having less than four processing units without deviating from the scope of the present invention. For example, if there are only two processing units, only two data lines will be pulled HIGH for arbitration, as will be described hereinbelow in reference to a exemplary generic embodiment.

The arbitration phase, which was designated as step **520** in FIG. **5**, is entered into with the start of a bus transaction shown in the step **601**. After the initialization step **602**, during the first arbitration cycle, the first four bus agents issue their respective bus request signals by asserting ACTIVE HIGH on their own bus request lines that are connected to the data lines of the bus **311** (shown in FIG. **3**). For example, during the first arbitration cycle, agent **0** which

is connected to OPDATA 0 (shown in FIG. 7), drives that line, referred to as DATA 0 in this flowchart. Similarly, agent 1 which is connected to OPDATA 1 (shown in FIG. 7), drives that line, referred to as DATA 1 in this flowchart; agent 2 which is connected to OPDATA 2 (shown in FIG. 7), drives that line, referred to as DATA 2 in this flowchart; and agent 3 which is connected to OPDATA 3 (shown in FIG. 7), drives that line, referred to as DATA 3 in this flowchart. If no more agents are present or detected, as provided in the step **604**, the master/arbiter COPIC will select a bus agent by employing a "rotating priority" or "round robin" arbitration protocol to grant the bus **311** for message delivery, as shown in steps **607** and **608**. For this purpose, the master/arbiter COPIC in the exemplary embodiment utilizes an internal circular pointer, denoted as ARB_PTR, that points to the bus agent that has been granted the most recent bus request. According to one aspect of the present invention, the bus agent that is pointed to by the contents of the ARB_PTR is accorded the least priority of bus arbitration; and, further, the bus agent that is pointed to by a wraparound increment of the ARB_PTR value by one is accorded the highest priority. In a presently preferred exemplary embodiment, a 5-bit ARB_PTR is provided for scalably accommodating up to 32 units, 31 bus agents and one master COPIC. On power-up, this 5-bit ARB_PTR is initialized to "11111" such that the bus agent pointed to by "00000" will have the highest priority for the up-coming bus arbitration. If, on the other hand, the total number of bus agents and master COPIC is less than 32, then, the most significant bits of the ARB_PTR will be appropriately masked.

Continuing to refer to FIG. **6**, once the control of the bus **311** is granted, the requester may immediately start sending its message, with a Cyclic Redundancy Check ("CRC") and ERROR check (shown in FIG. 7). After each successfully transmitted message, the master/arbiter COPIC increments or updates its ARB_PTR to point to the bus agent that successfully transmitted the message, as provided in step **609**. At the end of current bus transaction, the bus arbitration process phase may begin again for the next bus transaction **610**.

On the other hand, if more than four agents are arbitrating for bus control, then by taking the NO path from the decision block **604**, the arbitration cycle counter is incremented, as in step **606**, and the next arbitration cycle may start over again, with additional agents, for example, agents **4**, **5**, **6** and **7** driving data lines OPDATA **0**, OPDATA **1**, OPDATA **2**, and OPDATA **3**, respectively, as shown in FIG. **7**. If the list of agents is exhausted, as provided in **604**, then the bus control is granted in the same manner as discussed above.

From the foregoing, it can be readily appreciated by those skilled in the art that the present invention provides a highly cost-effective solution for balanced delivery of interrupts to their destinations in a scalable MP environment having four or more processing units without changing the bus width or bus protocol. The disclosed method requires that only CTPR values for each bus agent be compared for determining lowest current task priority, thereby ensuring that in the dynamic delivery mode, an interrupt is always distributed to the agent that is least busy. Further, by distributing control between local interrupt controllers and central interrupt controllers that are disposed on a dedicated interrupt bus, it can be realized that the impact on the host bus traffic or system bus traffic is rendered vanishingly small. With integration of local interrupt controllers with their corresponding processing units, a low cost, low pin count solution that is highly scalable in an MP system architecture having four or more processors is achieved. Furthermore, as described hereinabove, the present invention may also be practiced with fewer than four processing units without having to change the bus width or bus protocol. Accordingly, it is to be understood that the present invention provides a highly scalable solution that can support one or more processing units in an MP-environment.

Although a preferred embodiment of the method and apparatus of the present invention has been illustrated in the accompanying Drawings and described in the foregoing Detailed Description, it will be understood that the invention is not limited to the embodiment disclosed, but is capable of numerous rearrangements, modifications and substitutions without departing from the spirit of the invention as set forth and defined by the following claims.

What is claimed is:

1. A method for supporting multiple distributed interrupt controllers, designated as bus agents in a symmetric multiprocessing system, said method comprising the steps of:

assigning a unique identification number to each bus agent;

receiving bus requests from the bus agents in groups of four over a single programmable-interrupt-controller bus comprising four data lines;

arbitrating for control of said propgrammable-interrupt-controller bus, said arbitration step involving one or more arbitration phases each one of which phases corresponds to a particular group of four agents and is effectuated on said single programmable-interrupt-controller bus; and

granting bus ownership to a selected one of the requesting bus agents.

2. The method as set forth in claim **1**, wherein up to thirty-two bus agents can be supported without changing bus protocol or width associated with said single programmable-interrupt-controller bus.

3. A computer system that supports multiple distributed interrupt controllers, designated as bus agents, in a symmetric multiprocessing system, said computer system comprising:

a single programmable-interrupt-controller bus comprising four data lines;

means for assigning a unique identification number to each bus agent;

means for receiving bus requests from the bus agents in groups of four;

means for accommodating an arbitration scheme wherein an arbitration cycle comprises one or more arbitration phases, each one of which phases corresponds to a particular group of four bus agents and is effectuated on said single programmable-interrupt-controller bus; and

means for granting bus ownership to a selected one of the requesting bus agents.

4. The computer system as set forth in claim **3**, wherein up to thirty-two bus agents can be supported without changing bus protocol or width associated with said single programmable-interrupt-controller bus.

5. The computer system as recited in claim **3**, wherein said means for assigning comprises a central processor.

6. The computer system as recited in claim **3**, wherein said means for granting comprises a central programmable interrupt controller.

7. A method of arbitrating bus control requests in a computer system of the type including one or more processing units, each of which is in communication with a corresponding bus agent; and a master arbiter with an internal

arbitration pointer; the bus agents and the master arbiter disposed on a single programmable-interrupt-controller bus, the method comprising the steps of:

initializing the contents of the internal arbitration pointer;

receiving bus requests from the bus agents in groups of four over said single programmable-interrupt-controller bus comprising four data lines;

arbitrating for control of said single programmable-interrupt-controller bus, said arbitration step involving one or more arbitration phases, each one of which phases corresponds to a particular group of four agents; and

choosing a bus agent by the master arbiter based on an arbitration protocol, transmitting a bus message by the chosen bus agent, and updating the internal arbitration pointer to point to the chosen bus agent.

8. The method of arbitrating bus control requests in a computer system as set forth in claim **7**, further comprising the steps of:

initializing an arbitration cycle if more than four bus agents are listed; and

initializing a counter if more than four bus agents are listed.

9. The method of arbitrating bus control requests in a computer system as set forth in claim **8**, wherein said arbitration cycle is initialized by setting it to one and said counter is initialized by setting it to zero.

10. A computer system including a subsystem for arbitrating bus control requests of the type including one or more processing units, each of which is in communication with a corresponding bus agent, the bus agents disposed on a single programmable-interrupt-controller bus, the subsystem comprising:

means for initializing an internal arbitration pointer;

means for asserting a bus control request signal;

means for accommodating an arbitration scheme wherein an arbitration cycle comprises one or more arbitration phases, each one of which phases corresponds to a particular group of four bus agents and is effectuated on said single programmable-interrupt-controller bus;

means for choosing a listed bus agent based on said arbitration cycle; and

means for updating said internal arbitration pointer.

11. The computer system as set forth in claim **10**, wherein said means for asserting a bus control request comprises a signal generating means for generating an active high signal.

12. The computer system as set forth in claim **10**, wherein said means for choosing comprises a central programmable interrupt controller disposed on said single programmable-interrupt-controller bus.

* * * * *